Chombo Class PiecewiseLinearFillPatchFace

Dan Martin (code mods) and Gustav Meglicki (weave)

October 13, 2006

Prolongate.w,v 1.68 2006/10/13 22:05:20 gustav Exp

Contents

1 Copyright and Authors

Class **PiecewiseLinearFillPatchFace** discussed herein is based on Chombo's **PiecewiseLinearFillPatch**, code modifications by Dan Martin, LBNL, Friday, January 14, 2000.

This weave is by Gustav Meglicki, Indiana University, for the Argonne National Laboratory. It was prepared with CWEB-3.64 by Silvio Levy and Donald E. Knuth, and LaTeX classes RCS-2.10 by Joachim Schrod and Jeffrey Goldberg and CWEB-3.6 by Joachim Schrod.

2 Introduction

This document annotates and explains in painful detail class **PiecewiseLinearFillPatchFace** contributed to Chombo by Dan Martin. The class derives from a standard Chombo class **PiecewiseLinearFillPatch** and has been extended to work with a Chombo class **FluxBox** that is used to manage face centered data such as fluxes.

For our nano-photonics project and FDTD computations we must ensure that data that has been "prolongated" on the fine grid faces is divergence free. Now, **PiecewiseLinearFillPatchFace** does not do this, but by analyzing how and what it does, we should be able to produce the required remedy, for example, as a *change* file to this document, or as an overload of currently available class methods.

Class **PiecewiseLinearFillPatchFace** is central to SHAPES. Because media distribution in SHAPES is static, we have little need for moving Chombo grids, and we won't in the future 3D version of the code. Once the data on the grids have been initialized at the beginning of the computation to zero, we never have to reconstruct fields on the fine grids again. Field restriction, i.e., transfer from a fine to a coarse level, is trivial and follows Balsara's and Dan's procedure of averaging fine data over the coarse grid faces, while ignoring the data that's between the coarse grid faces. This simple procedure transfers divergence free property from the finer to the coarser level. We may have to add a divergence killing correction on the fine/coarse boundary. This is the *refluxing* part of the procedure, and it is going to be analyzed in a separate document.

But fine-grid boundary updates at the fine/coarse boundaries have to be carried out at every time step and it is here that we had observed instabilities arising, especially near the fine region corners.

Consequently, this part of the code must be scrutinized and brought under our full control. Luckily, the code is written entirely in C++, and without any references to Chombo Fortran stubs, which makes its analysis somewhat easier.

This document comprises two large sections. The first section, 3, is the header file. It introduces data structures and methods of the class. Then the second section, 4, introduces the code itself.

The second section is subdivided into two major subsections. The first one, 4.1, is concerned with defining the border between a fine and a coarse region and allocating various data structures used in calculating the prolongation. This is the role of function *define*, which is a *de facto* constructor of the class. The second subsection, 4.3, fills the coarse/fine border on the fine grid side with data obtained from the coarse grid. It prolongates the data.

The prolongation implemented in **PiecewiseLinearFillPatchFace** is just simple linear interpolation with slope limiters. This is a good starting point on the way to divergence-free Balsara prolongation. Once we get to understand how this code handles the data and how the actual computations are done, we should be able to expand on the procedures as needed.

Throughout the weave problematic parts of the code are flagged with the Knuth dangerous bend sign, as shown in the margin of this paragraph.

The general algorithm presented here is fairly simple. We start with data on a coarse grid and on a fine grid. First we create a *coarsened* fine grid, i.e., a grid that is coarse, but that overlaps exactly with the fine grid. It is a coarse-world image of the fine grid. Then we create **LevelData** on this *coarsened* fine grid. At this stage we give it a border belt of ghost cells which is wide enough to cover entirely the border belt of the fine grid ghost cells and then stretch a little more. Then we copy data from the original coarse grid to this *coarsened* fine grid. This operation fills the ghost cells of the *coarsened* fine grid as well.

The border cells of the fine and the *coarsened* fine grids are identified by swelling each box of the grid and then subtracting all other boxes of the grid from it. If the swollen box is in the middle of the grid, this procedure subtracts all its points and produces an empty set. But if the box is on the boundary of the grid, then the cells that stick out aren't subtracted and these are the cells that identify the boundary. These cells are also ghost cells of both the fine and the *coarsened* fine grids.

Once the *coarsened* fine grid border cells are identified, slopes can be computed on them and these are then used in interpolating data onto the fine grid border cells.

Having to deal with face centered data, periodic boundaries, slope and face directions, etc., introduces various complications that have to dealt with.

3 The Header File

The content of the header file is enclosed in the # **ifndef** clause that ensures that its definitions won't be read more than once into the compilation stream, even if the file is included several times. The name of the class is **PiecewiseLinearFillPatchFace** and its purpose is to fill coarse/fine border ghost cells on the fine sub-grid side with data generated from coarse grid data. In the case of this class, as we have remarked earlier, the data is linearly interpolated.

The class constructs the boundary region when it is first defined. This is not a cheap operation and so, if a given subgrid is going to remain static throughout the computation, the class should be defined just once.

The class does not define any private interfaces or variables. Its *protected* interfaces and variables may be accessed by *derived* classes, as if they were public, but cannot be accessed by the rest of the program—this is the standard C++ meaning of the keyword **protected**.

3 ⟨ Prolongate.H 3 ⟩ ≡

```
#ifndef _PIECEWISE_LINEAR_FILL_PATCH_FACE_H_
# define _PIECEWISE_LINEAR_FILL_PATCH_FACE_H_
  ⟨ Includes 4 ⟩
  class PiecewiseLinearFillPatchFace {
    ⟨ Public Interfaces 5 ⟩
    ⟨ Protected Interfaces 6 ⟩
    ⟨ Protected Variables 7 ⟩
  }
#endif
```

¶ The includes are as follows. First we include standard C++ *iostream* and *fstream*, which define IO. Then we have Chombo specific includes that define:

`REAL.H` This file defines the meaning of the Chombo term **Real**, which may be a double or a single precision floating point number depending on how the Chombo library has been configured.

`Box.H` This file defines the **Box** class, which is a rectangular box of grid points (cells). To be more specific, boxes can be defined to support various types of data placement, e.g., cell centered, face centered, etc.

`FArrayBox.H` This file defines the **FArrayBox** class, which is a field of n **Real** components defined on a **Box**.

`FluxBox.H` This file defines the **FluxBox** class, which has a separate **FArrayBox** field associated with each face of a grid cell. This class may be thought of as wrapping three (in 3D) face-mounted or two (in 2D) **FArrayBox**es into a single object. This class supports *setVal*, *copy* and *shift* operations, the same way **FArrayBox** does, and **LevelData** (see the next item) of **FluxBox**es can be constructed the same way as **LevelData** of **FArrayBox**es.

`LevelData.H` This file defines the **LevelData** class template, which associates a field with a subgrid. The subgrid is divided into **Box**es, which must form a **DisjointBoxLayout**. The field is either an **FArrayBox** defined over every **Box** of the layout, or a bundle of these, that is a **FluxBox**, defined over every **Box** of the layout. But **LevelData** may also associate other types of data with every **Box** of the layout, for example sets of grid points.

`IntVectSet.H` This file defines the **IntVectSet** class, which is a set of **IntVect**s, which are *integer vectors.* These normally define grid cells. A set of these is therefore a set of grid cells—but we will also refer to them as *points.* We are going to represent a border between a coarse and a fine level by a **LevelData** of grid cell sets, where the cells form the border. In other words, each **Box** of the **DisjointBoxLayout** will have a set of border points associated with it. If a given **Box** does not abut the border, the set will be empty.

`ProblemDomain.H` This file defines the **ProblemDomain** class, which can be thought of as a large **Box** that encloses all the **Box**es of all **DisjointBoxLayout**s at all levels. The sides of a **ProblemDomain** is where boundary conditions for the computation, periodic or otherwise, are applied, and a

ProblemDomain carries *some* information about it—periodic or non-periodic, if periodic, in which direction. This is the main difference between it and a **Box**. In most Chombo function calls a **Box** may be used in place of a **ProblemDomain**, in which case the domain is assumed non-periodic.

4 ⟨ Includes 4 ⟩ ≡
```
#include <iostream>
#include <fstream>
#include "REAL.H"
#include "Box.H"
#include "FArrayBox.H"
#include "FluxBox.H"
#include "LevelData.H"
#include "IntVectSet.H"
#include "ProblemDomain.H"
```
This code is used in chunk 3.

¶ The public interfaces declared in the header file are for the creator and destructor of the class, which can be called without any or with some arguments, about which more below. We also have a method *define*(), which is used by the class to build required data structures and make an actual object. Then there is a small boolean function *isDefined*( ) that doesn't take any arguments and that is discussed in chunk ⟨ Inquiry functions 24 ⟩, page 21, and an auxiliary debugging function *printIntVectSets*( ) that doesn't take any arguments either and that is discussed in chunk ⟨ Debugging Utilities 39 ⟩, page 33.

Once the class has been defined, a function *fillInterp* can be called that fills coarse/fine border cells on the fine grid side with coarse grid data that have been time and space interpolated and additionally massaged in various ways (van Leer slope limiting). This stuff is discussed in chunks ⟨ Fill the Border 25 ⟩, page 22, to ⟨ Interpolate between faces 38 ⟩, page 31.

Functions *isDefined*( ) and *printIntVectSets*( ) are declared to be *inspectors,* as opposed to *modifiers,* meaning that they don't change anything inside the class. This is the meaning of the word **const** that follows the interface definition of the method. The functions are used to tell us if the class has been fully defined and, if it has, to print the list of coarse/fine border cells.

The arguments needed to define the class, and used by both *define* and the **PiecewiseLinearFillPatchFace** constructor, are as follows

a_fine_domain This is the **DisjointBoxLayout** of the fine grid. This argument is passed by reference, and the reference is not changed (it is **const**) by the constructor.

a_coarse_domain This is the **DisjointBoxLayout** of the coarse grid. This argument is also passed by reference, and the reference is not changed (it is **const**) by the constructor.

a_num_comps The number of field components... per face. We must remember here that the way a vector field, e.g., $\vec{B}$ is encoded in a **FluxBox** is that each component of $\vec{B}$ is associated with a different face of the box cell. And so a **FluxBox** associates three separate 1-component **FArrayBox**es with each box of nodes. The value of *a_num_comps* in this case is 1, not 3. But, say, if the field was such that there would be a tri-vector associated with each face of the box, then *a_num_comps* would be 3. An example of such a field could be a 3×3 tensor field.

a_crse_problem_domain This can be either a **ProblemDomain** or a **Box** that encloses the coarse domain grid and overlaps with the whole computational domain of the problem, sharing boundary periodicities with it, if any. It has the same resolution as *a_coarse_domain*. If a **Box** is used in this place, it is internally upgraded to a non-periodic **ProblemDomain** that is built around the box.

a_ref_ratio A refinement ratio between coarse and fine grids.

a_interp_radius An interpolation radius, i.e., the radius in fine grid cells, of the coarse grid region, that is scanned to produce the interpolation. This number must be an integral multiple of the refinement ratio, i.e., if *a_ref_ratio* is 2, then *a_interp_radius* must be 2, 4, 6, etc.,

Function *fillInterp* takes the following arguments:

a_fine_data This is the field that's defined on the fine grid.

a_old_coarse_data This is the "old" version of the field that's defined on the coarse grid. The data is going to be first time-interpolated between the old coarse data and the new coarse data, and then only it will be space-interpolated. See chunk ⟨ Fill the Border 25 ⟩, page 22.

a_new_coarse_data This is the "new" version of the field that's defined on the coarse grid.

a_time_interp_coef This is the time interpolation coefficient. It has to be a real number between 0.0 and 1.0. 0.0 means "take just the old data", 1.0 means "take just the new data", and anything between 0.0 and 1.0 delivers a mixture of old and new. See the next chunk, ⟨ Protected Interfaces 6 ⟩, and chunk ⟨ Time Interpolation 27 ⟩, that discusses the internals of the time interpolating function, for more details.

a_src_comp This is the first component on the coarse grid side from which to start the interpolation.

a_dest_comp This is the first component on the fine grid side into which to start the writing.

a_num_comp This is the number of components to be interpolated and transferred into the coarse/fine border cells on the fine grid side.

The same comments apply to these components as above, i.e., a vector field $\vec{B}$ is a 1-component field, because there is one component associated with each box face. But if there is more than one component associated with each face, then we also have to specify how many components we want to interpolate and this number is *a_num_comp*. The interpolated components may not necessarily be written onto the same components on the destination side. The user may wish to write them on different components. In this case the first destination component must be provided in *a_dest_comp*.

5 ⟨ Public Interfaces 5 ⟩ ≡

```
public:
  PiecewiseLinearFillPatchFace( );
  ~PiecewiseLinearFillPatchFace( );
  PiecewiseLinearFillPatchFace(const DisjointBoxLayout &a_fine_domain, const DisjointBoxLayout
      &a_coarse_domain, int a_num_comps, const Box &a_crse_problem_domain, int a_ref_ratio, int
      a_interp_radius);
  PiecewiseLinearFillPatchFace(const DisjointBoxLayout &a_fine_domain, const DisjointBoxLayout
      &a_coarse_domain, int a_num_comps, const ProblemDomain &a_crse_problem_domain, int
      a_ref_ratio, int a_interp_radius);
  void define(const DisjointBoxLayout &a_fine_domain, const DisjointBoxLayout &a_coarse_domain, int
      a_num_comps, const Box &a_crse_problem_domain, int a_ref_ratio, int a_interp_radius);
  void define(const DisjointBoxLayout &a_fine_domain, const DisjointBoxLayout &a_coarse_domain, int
      a_num_comps, const ProblemDomain &a_crse_problem_domain, int a_ref_ratio, int a_interp_radius);
  bool isDefined( ) const;
  void fillInterp(LevelData⟨FluxBox⟩ &a_fine_data, const LevelData⟨FluxBox⟩ &a_old_coarse_data, const
      LevelData⟨FluxBox⟩ &a_new_coarse_data, Real a_time_interp_coef, int a_src_comp, int
      a_dest_comp, int a_num_comp);
  void printIntVectSets( ) const;
```

This code is cited in chunks 10, 11, and 25.

This code is used in chunk 3.

¶ In this chunk we define protected function interfaces of the class. The word **protected** here means that they're private to the class, but may be accessed by the class heirs, as opposed to data specified as **private**, which would mean that only *the* class may access them. This is done so that the class may be effectively extended in future.

These functions are all internal and not visible to the user of the class. The functions declared here are

timeInterp This **protected** function performs linear time interpolation between *a_old_coarse_data* and *a_new_coarse_data* where *a_time_interp_coef* specifies the point of interpolation: 0.0 means interpolate at the *a_old_coarse_data* time, 1.0 means interpolate at the *a_new_coarse_data* time. This **Real** number may be anything between 0 and 1. The function interpolates multiple components beginning with *a_src_comp*. The number of components to interpolate is *a_num_comp* and the interpolated data is written on the internal storage of the class, *m_coarsened_fine_data*, defined in the next chunk, on destination components beginning with *a_dest_comp*. The function is discussed in chunk ⟨ Time Interpolation 27 ⟩, page 23.

fillConstantInterp This **protected** function transfers field values that live in a given cell of the *coarsened* fine grid, i.e., in *m_coarsened_fine_data*, to the fine grid cells it overlaps with. The data is written on *a_fine_data*. The field values are then further tweaked to produce slope limited interpolations. The arguments to this function are *a_fine_data*, the first component on the source side, the first component on the destination side, and the number of components to be transferred. This function is discussed in chunk ⟨ Piecewise Constant Interpolation 29 ⟩, page 24.

computeSlopes This **protected** function computes slopes of a field stored in *m_coarsened_fine_data* in the direction *a_dir*. The slopes for a field associated with a given face direction are computed within the face only. The slopes are limited so as to avoid generation of artifacts. The slope directions are: 0 for x, 1 for y and 2 for z. Once evaluated, the slopes will be used by function *incrementLinearInterpTangential* in the interpolation of data on the fine grid faces that overlap with faces of the *coarsened* fine grid. The slopes are calculated for *a_num_comp* components beginning with *a_src_comp* and stored in component slots that begin with *a_dest_comp* on the destination side. This function is discussed in chunk ⟨ Evaluation of Slopes 31 ⟩, page 26.

incrementLinearInterpTangential This **protected** function implements linear corrections in the direction *a_dir* to field values on fine grid faces that overlap with faces of the *coarsened* fine grid using van Leer limited slopes evaluated by *computeSlopes*. It is discussed in chunk ⟨ Tangent Correction 35 ⟩, page 29. Its arguments are the fine grid data, *a_fine_data*, the slope direction, *a_dir*, source and destination components *a_src_comp* and *a_dest_comp*, and the number of components to interpolate.

incrementLinearInterpNormal This **protected** function, which is discussed in chunk ⟨ Normal Correction 37 ⟩, page 30, interpolates data on fine grid faces that do not overlap with *coarsened* fine grid faces, i.e., it fills the interior fine grid faces, interior with respect to the *coarsened* fine grid. It does so by simple linear interpolation between the fine grid data that live on the faces shared with the *coarsened* fine grid. The arguments are *a_fine_data*, which is the fine grid data, a number of field components to interpolate, *a_num_comp*, the first component on the source side, *a_src_comp*, and the first component on the destination side, *a_dest_comp*.

Functions that do not affect internal variables are the *inspectors,* but they may write on external data layouts. The functions flagged with **const** in the listing below are in this category.

6 ⟨ Protected Interfaces 6 ⟩ ≡

```
protected:
  void timeInterp(const LevelData<FluxBox> &a_old_coarse_data, const LevelData<FluxBox>
      &a_new_coarse_data, Real a_time_interp_coef, int a_src_comp, int a_dest_comp, int a_num_comp);
  void fillConstantInterp(LevelData<FluxBox> &a_fine_data, int a_src_comp, int a_dest_comp, int
      a_num_comp) const;
  void computeSlopes(int a_dir, int a_src_comp, int a_num_comp);
  void incrementLinearInterpTangential(LevelData<FluxBox> &a_fine_data, int a_dir, int a_src_comp, int
      a_dest_comp, int a_num_comp) const;
  void incrementLinearInterpNormal(LevelData<FluxBox> &a_fine_data, int a_src_comp, int a_dest_comp, int
      a_num_comp) const;
```

This code is cited in chunk 5.

This code is used in chunk 3.

¶ And finally we have internal **protected** variables, namely

m_is_defined This parameter is set to *true* when the object of class **PiecewiseLinearFillPatchFace** is fully defined. Other class functions look it up, to check if it's safe to operate on the data.

s_stencil_radius This is a constant parameter that is used in just one place in function *define*, when calculating the *coarse_ghost_radius*, i.e., the number of ghost cells that need to be added to the *coarsened* fine grid boxes (see *m_coarsened_fine_data* below for the discussion), in chunk ⟨ create private data structures 14 ⟩, page 12. This is the distance that we need to go outside the fine grid boundary region to collect data for centered differences. See chunk ⟨ create private data structures 14 ⟩, page 12, for a more detailed discussion. It is set to 1 in chunk ⟨ CPP File Includes 9 ⟩, page 9.

m_ref_ratio This is an internal copy of the refinement ratio.

m_interp_radius This is an internal copy of the interpolation radius.

m_coarsened_fine_data This is a data structure that corresponds to the fine level data, but it's constructed on a new coarse sub-grid that overlaps exactly with the fine grid. The coarse sub-grid is obtained by *coarsen*-ing the fine grid. The data from the original coarse level grid is first copied onto *m_coarsened_fine_data*, and then all following computations—slopes—are calculated on *m_coarsened_fine_data*. Some data is copied from *m_coarsened_fine_data* onto *a_fine_data* by function *fillConstantInterp* discussed in chunk ⟨ Piecewise Constant Interpolation 29 ⟩, page 24.

m_slopes This is a data structure that lives on the same grid as *m_coarsened_fine_data*—also a **FluxBox**—but contains limited slopes of field values. It is used in calculating linear corrections to piecewise constant interpolation in chunk ⟨ Tangent Correction 35 ⟩, page 29. This field is re-used for different slope directions as a temporary store, because as soon as the slopes for a given direction have been computed and stored on it, they are used by the next function call, so they can be overwritten in the next iteration over slope directions.

m_crse_problem_domain This is an internal copy of the coarse level problem domain.

The following four variables are layouts of sets. They are constructed by associating a set of grid points (**IntVect**s) with each box of the layout. The coarse/fine grid border is defined in terms of these. If a given box is internal and does not abut a border, its corresponding set is going to be empty. But if the box abuts the border, then the nodes of the box that are on the border go into the set. We can then iterate, first over the boxes of the layout and then over the points of the set associated with each box, to perform computations on data associated with boundary nodes.

The first of the four set layouts, *m_fine_interp*, is associated with the fine grid box layout and the remaining three are associated with the *coarsened* fine grid box layout. The coarsened sets overlap with the fine sets, but they stretch a little beyond them by the length of the stencil radius and then still a little bit.

m_fine_interp This is an array of set layouts, one for each face direction, that contains points onto which data will be interpolated. In the original Chombo **PiecewiseLinearFillPatch** code this variable is not an array. But here we have separate **FArrayBox**es associated with each face of each cell. For each of the face directions we collect the coarse/fine border points separately into the corresponding component of the array. The boxes of the layout in this case are those of the fine grid.

m_coarse_centered_interp This is a matrix of set layouts. The second index numbers cell faces, as is the case with *m_fine_interp* above. The first index numbers the directions in which slopes are calculated—and they will be calculated only in the directions that lie within the face, i.e., that are perpendicular to the direction of the face. The box layouts here are those of the *coarsened* fine grid. For most points slopes can be evaluated from both sides, because we grow the border belt on the *coarsened* fine grid side sufficiently wide to incorporate all points that are needed (this is what the stencil radius constant is for). However, if in some locations the coarse grid outer boundary, i.e., the coarse/coarser boundary, gets so close to the fine grid outer boundary, i.e., the fine/coarse boundary, that this cannot be done, then we have to resort to the evaluation of one-sided differences. This is a somewhat pathological situation and it is possible to make the code flag a problem if this happens. We will collect such points on the sets defined below, *m_coarse_lo_interp* and *m_coarse_hi_interp*. This set layout, *m_coarse_centered_interp* , is for the points, for which we can evaluate central differences.

m_coarse_lo_interp This set layout is for the border points, for which we will evaluate one-sided differences "from the left hand side", otherwise it is like *m_coarse_centered_interp*. The lookup to the left means that the point is at the right edge of our computational space, because only then we are guaranteed to find data to the left of it.

m_coarse_hi_interp This set layout is for the border points, for which we are going to evaluate one-sided differences "from the right hand side", otherwise it is like *m_coarse_centered_interp*. The lookup to the right means that the point is at the left edge of our computational space, because only then we are guaranteed to find data to the right of it.

The slopes for a given face and slope direction, by whichever means they are obtained, are written on *m_slopes*. *m_slopes* no longer knows if given slopes come from central differences or one-sided differences.

7 ⟨ Protected Variables 7 ⟩ ≡

```
protected:
  bool m_is_defined;
  static const int s_stencil_radius;
  int m_ref_ratio;
  int m_interp_radius;
  LevelData<FluxBox> m_coarsened_fine_data;
  LevelData<FluxBox> m_slopes;
  ProblemDomain m_crse_problem_domain;
  LayoutData<IntVectSet> m_fine_interp[SpaceDim];
  LayoutData<IntVectSet> m_coarse_centered_interp[SpaceDim][SpaceDim];
  LayoutData<IntVectSet> m_coarse_lo_interp[SpaceDim][SpaceDim];
  LayoutData<IntVectSet> m_coarse_hi_interp[SpaceDim][SpaceDim];
```

This code is cited in chunks 9, 11, 16, and 20.

This code is used in chunk 3.

4 The CPP File

The CPP file includes the header file, described in the previous sections, and other Chombo and C++ headers, then lists various class construction wrappers. Eventually we get to *define,* which constructs the *coarsened* fine grid and other private data structures, and identifies the fine/coarse boundary points.

Function *fillInterp* is a simple wrapper around five auxiliary **private** functions that do the actual job of interpolating data and writing them on the appropriate locations in the fine grid's boundary. The auxiliary functions are then defined following *fillInterp*.

Finally, there is a debugging utility *printIntVectSets* that can be used to print a list of boundary nodes found by *define.*

8 ⟨ CPP File Includes 9 ⟩
⟨ Wrappers for Define 10 ⟩
⟨ Define the Border 11 ⟩
⟨ Inquiry functions 24 ⟩
⟨ Fill the Border 25 ⟩
⟨ Debugging Utilities 39 ⟩

¶ Some of the includes here, like `"REAL.H"`, `"Box.H"`, `"FArrayBox.H"`, `"LevelData.H"`, and `"IntVectSet.H"` overlap with includes in the header file and so they're not really needed. The Chombo includes that are new are

`IntVect.H` This file defines **IntVect**s themselves, i.e., integer vectors that describe grid points (cells).

`DisjointBoxLayout.H` This file defines a layout of boxes of grid points that constitutes a grid *level.* The boxes may be distributed over multiple CPUs, but Chombo handles parallelism quite transparently. Seldom do we have to do anything about it explicitly.

`LayoutIterator.H` This file defines a device that is used to iterate over boxes of a given layout.

`MayDay.H` This file defines a device for flagging errors and aborting the program.

We also include `"cmath"`, which is a part of the standard C++ library that covers mathematics. The two **using** lines tell the compiler that *cout* and *endl*, if encountered in the program text, should be picked up from the *std* package.

Finally, there is a little **inline** definition of a utility, called *copysign*, that transfers a sign from its second argument to its first argument. This utility is employed in one place only, in chunk ⟨ Evaluate van Leer limited central differences 32 ⟩, page 27.

At the end of this section we fix the *s_stencil_radius* constant, discussed in chunk ⟨ Protected Variables 7 ⟩, page 6, at 1. As we have mentioned before, this variable is used in one place only in chunk ⟨ create private data structures 14 ⟩, page 12, where its meaning is discussed in more detail. Setting it at 1 means that we restrict ourselves to linear interpolations based on centered differences.

9 ⟨ CPP File Includes 9 ⟩ ≡

```
#include <cmath>
#include "REAL.H"
#include "IntVect.H"
#include "Box.H"
#include "FArrayBox.H"
#include "LevelData.H"
#include "IntVectSet.H"
#include "DisjointBoxLayout.H"
#include "LayoutIterator.H"
#include "MayDay.H"
  using std::cout;
  using std::endl;
#include "PiecewiseLinearFillPatchFace.H"
#ifndef copysign
```

```
  template⟨class T⟩ inline T copysign(const T&a, const T&b)
  {
    return (b ≥ 0) ? ((a ≥ 0) ? a : −a) : ((a ≥ 0) ? −a : a);
  }
#endif
  const int PiecewiseLinearFillPatchFace::s_stencil_radius = 1;
```

This code is cited in chunks 7 and 14.

This code is used in chunk 8.

¶ This chunk introduces various wrappers around the actual utility that constructs the class. The real constructor is the class method *define* discussed in the next chunk.

The constructor itself may be called without any arguments, in which case nothing is constructed and the protected class variable *m_is_defined* is set to *false*.

The destructor, ∼**PiecewiseLinearFillPatchFace**( ), doesn't do anything. There is no explicit garbage collection here—though other classes that are invoked by this one, e.g., **LevelData**, may attend to their own clean-up.

As we have discussed in chunk ⟨ Public Interfaces 5 ⟩ the constructor may be invoked with a coarse problem domain specified either as a **Box** or as a **ProblemDomain**. If the domain is specified as a **Box**, the wrapper makes it into a non-periodic **ProblemDomain**, and then calls *define*. Otherwise, the wrapper calls *define* and passes all arguments it has received to it without change. Both wrappers set *m_is_defined* to *false*. It is then up to *define* to replace this with *true*, once the object has been fully constructed.

Function *define* similarly may be called with the coarse problem domain defined as a **Box**, in which case the **Box** is converted to a non-periodic **ProblemDomain** and the function then calls its other instantiation that carries out the object construction.

10 ⟨ Wrappers for Define 10 ⟩ ≡

```
  PiecewiseLinearFillPatchFace::PiecewiseLinearFillPatchFace( ) : m_is_defined(false)
  { }
  PiecewiseLinearFillPatchFace::∼PiecewiseLinearFillPatchFace( )
  { }
  PiecewiseLinearFillPatchFace::PiecewiseLinearFillPatchFace(const DisjointBoxLayout
      &a_fine_domain, const DisjointBoxLayout &a_coarse_domain, int a_num_comps, const Box
      &a_crse_problem_domain, int a_ref_ratio, int a_interp_radius) : m_is_defined(false)
  {
    ProblemDomain crsephysdomain(a_crse_problem_domain);
    define(a_fine_domain, a_coarse_domain, a_num_comps, crsephysdomain, a_ref_ratio, a_interp_radius);
  }
  PiecewiseLinearFillPatchFace::PiecewiseLinearFillPatchFace(const DisjointBoxLayout
      &a_fine_domain, const DisjointBoxLayout &a_coarse_domain, int a_num_comps, const
      ProblemDomain &a_crse_problem_domain, int a_ref_ratio, int a_interp_radius) : m_is_defined(false)
  {
    define(a_fine_domain, a_coarse_domain, a_num_comps, a_crse_problem_domain, a_ref_ratio, a_interp_radius);
  }
  void PiecewiseLinearFillPatchFace::define(const DisjointBoxLayout &a_fine_domain, const
          DisjointBoxLayout &a_coarse_domain, int a_num_comps, const Box &a_crse_problem_domain, int
          a_ref_ratio, int a_interp_radius)
  {
    ProblemDomain crsephysdomain(a_crse_problem_domain);
    define(a_fine_domain, a_coarse_domain, a_num_comps, crsephysdomain, a_ref_ratio, a_interp_radius);
  }
```

This code is used in chunk 8.

4.1 Defining the Border

The class method *define*, creates data structures that are needed to characterize and *wrap* the coarse/fine border and to carry out required interpolations. Also, it identifies the coarse/fine border itself and stores this information. The data structures and the definition of the border are then used by function *fillInterp*, see chunk⟨ Fill the Border 25 ⟩, page 22, that fills the fine level border ghost cells with data obtained from the coarse level.

The calling parameters for *define* are as we have discussed in chunk ⟨ Public Interfaces 5 ⟩, page 4.

The function itself is rather long and complicated, but its general outline is fairly simple. First, it transfers data to the simple **protected** class variables *m_ref_ratio*, *m_interp_radius*, and *m_coarse_problem_domain*, already discussed in chunk ⟨ Protected Variables 7 ⟩, page 6, and performs some basic sanity checks.

Then it checks if the coarse level box layout, *a_coarse_domain*, is fully defined. When a layout definition is completed, the layout gets *closed*. Closing a box layout sorts the boxes and makes them available to other Chombo operations, such as writing data on them. A function that checks if the layout has been closed is called *isClosed*( ).

So, if the coarse level box layout has been closed, then only does *define* perform other operations, and if it gets safely to the end, it sets *m_is_defined* to *true* and returns. Come to think of it, we should check if *a_fine_domain* has been closed, too, but we don't. Is this an omission?

The *other* operations mentioned aboved are

1. creation of the protected data structures *m_slopes* and *m_coarsened_fine_data*—already discussed in chunk ⟨ Protected Variables 7 ⟩, page 6, and
2. a loop over the face directions, where most of the action takes place.

Recall that we are dealing with face-mounted fields here. For each face direction then we are going to perform all the operations that **PiecewiseLinearFillPatch**::*define* does once only, so that every one of these face-centered fields is taken care of, separately.

11 ⟨ Define the Border 11 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::define(const DisjointBoxLayout &a_fine_domain, const
        DisjointBoxLayout &a_coarse_domain, int a_num_comps, const ProblemDomain
        &a_crse_problem_domain, int a_ref_ratio, int a_interp_radius)
{
  ⟨ transfer data to private variables 12 ⟩
  ⟨ perform sanity checks 13 ⟩
  if (a_coarse_domain.isClosed( )) {
    ⟨ create private data structures 14 ⟩
    ⟨ loop over face directions 15 ⟩
    m_is_defined = true;
  }
}
```

This code is cited in chunks 14 and 15.

This code is used in chunk 8.

¶ This is a trivial chunk that transfers input data to the class protected variables *m_ref_ratio*, *m_interp_radius* and *m_crse_problem_domain*.

Tacked on to this chunk is also a definition of a shift iterator associated with the **ProblemDomain** *m_crse_problem_domain*. The **ShiftIterator** class contains a list of shift vectors that are used to enforce periodic boundary conditions, if such are present. This definition doesn't really have to be here, but is. It is used in chunks ⟨ Make correction for periodic boundary conditions 20 ⟩, ⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩ and ⟨ Collect fine cells for interpolation 23 ⟩.

12 ⟨ transfer data to private variables 12 ⟩ ≡

```
m_ref_ratio = a_ref_ratio;
m_interp_radius = a_interp_radius;
m_crse_problem_domain = a_crse_problem_domain;
```

ShiftIterator *shiftIt* = *m_crse_problem_domain.shiftIterator*( );

This code is cited in chunk 20.

This code is used in chunk 11.

¶ The sanity checks in this chunk are not exactly comprehensive.

First, we check if *a_interp_radius* is a multiple of *a_ref_ratio*. The reason why we want this is because we will interpolate over whole cells of the *coarsened* fine grid, whereas *a_interp_radius* is given in terms of the fine grid constant. The width of the *coarsened* fine grid border belt to scan data from for interpolation will be *a_interp_radius*/*a_ref_ratio* plus an additional reach that will be discussed in more detail in chunk ⟨ create private data structures 14 ⟩, page 12, where the actual sizing and shaping of the *m_slopes* and *m_coarsened_fine_data* grids happens.

If *a_interp_radius* is not a multiple of *a_ref_ratio*, we print an error message on *cerr* and exit via **MayDay**::*Abort*( ). This is what **MayDay**::*Error*( ) does.

The code could be a little more friendly here and automatic adjustment of *m_interp_radius* to the multiple of *a_ref_ratio* from a given *a_interp_radius* could be implemented easily. Also, the code does not check if the interpolation radius is large enough to cover all ghost cells of the fine grid. The user must ensure this by matching one against the other.

Then we check if *a_fine_domain* has the same periodicity as *m_crse_problem_domain*, which is by now a copy of *a_crse_problem_domain*. This is done by refining the coarse domain into what we call just here *fine_problem_domain*, and comparing it against *a_fine_domain* that's been passed to the function through the argument list.

a_fine_domain is a disjoint box layout of the fine level. The **DisjointBoxLayout** method *checkPeriodic*, which takes a **ProblemDomain** as its argument checks if its box layout is compatible with the domain. To be compatible both must have the same periodicity in all directions and with the same periods. This is also why we could not use *m_crse_problem_domain* in this check, because then the periods would be different. We had to create a refined version that would match the grid spacing of the fine level.

The C NewLib macro *assert* turns into **void** if the program is compiled with the −DNDEBUG flag, which is a default for Chombo. To activate *assert*s, a DEBUG version of the library must be used.

Normally, when *assert* receives *false* it prints a message showing what failed and where and aborts.

13 ⟨ perform sanity checks 13 ⟩ ≡

```
if (a_interp_radius ≠ (a_interp_radius/a_ref_ratio) * a_ref_ratio) {
  MayDay::Error("PiecewiseLinearFillPatchFace::define: interp_radius must be\
      integral multiple of nRef");
}
const ProblemDomain fine_problem_domain = refine(m_crse_problem_domain, m_ref_ratio);
assert(a_fine_domain.checkPeriodic(fine_problem_domain));
```

This code is cited in chunk 14.

This code is used in chunk 11.

¶ Now we enter the all-embracing **if** (*a_coarse_domain.isClosed*( )) statement of chunk ⟨ Define the Border 11 ⟩.

First, we check if the **DisjointBoxLayout** *a_coarse_domain* is compatible with the **ProblemDomain** *a_crse_problem_domain*, the same way we did it in chunk ⟨ perform sanity checks 13 ⟩ for the **DisjointBoxLayout** *a_fine_domain*.

Then we get down to the business of building *m_coarsened_fine_data* and *m_slopes*. Both fields will live on a grid that is a *coarsened* copy of *a_fine_domain*. To make this grid we define it first and then instantiate to the coarsened version of *a_fine_domain* by calling Chombo function *coarsen*.

In order to call **LevelData**⟨**FluxBox**⟩::*define* on both *m_slopes* and *m_coarsened_fine_data* we still have to decide on the width of ghost cell margins for both fields, and it is here that we use *s_stencil_radius* that was introduced so mysteriously in chunk⟨ CPP File Includes 9 ⟩, page 9.

The formula for both fields is roughly speaking as we have already pointed out in chunk ⟨ perform sanity checks 13 ⟩, page 12, i.e., *m_interp_radius* / *m_ref_ratio* plus an additional reach to ensure the availability of points for centered differences.

The extra reach for *m_slopes* does not reach anywhere. The formula is

$$\frac{nr + r - 1}{r} = n + \frac{r - 1}{r},$$

where nr is *m_interp_radius* (in multiples of *m_ref_ratio*), and r is *m_ref_ratio*. Because it is all done within the integer arithmetic, the term $(r-1)/r$ always truncates to zero. So, in effect we end up with *coarse_slope_radius* being simply n. The formula used in the code probably derives from the days before the enforcement for *m_interp_radius* has been put in chunk ⟨ perform sanity checks 13 ⟩ and is currently redundant, i.e., it can be replaced simply with *m_interp_radius* / *m_ref_ratio*.

The *coarse_ghost_radius* is this plus 2, because *s_stencil_radius* has been set to 1. Now, what is this *coarse_ghost_radius*. It is the number of ghost cells that are going to be added to *m_coarsened_fine_data* boxes when it is created. This is *m_interp_radius* / *m_ref_ratio*, which is how far we will grow the coarse/fine boundary belt on the fine grid side in chunk ⟨ Collect coarse cells for interpolation 19 ⟩, page 16, and incidentally also how far the *m_slopes* boxes reach, plus an additional *stretch* to collect data for centered differences. This additional stretch is *s_stencil_radius*. To be on the safe side though, we still add one more cell, sic!

In summary, we are going to surround *m_coarsened_fine_data* grid boxes with a sufficiently thick layer of ghost cells so that the actual coarse/fine border cells we'll identify in ⟨ Collect coarse cells for interpolation 19 ⟩ should cover entirely the *m_slopes* and *a_fine_data* border belts with a sufficient additional margin to find points for centered differences at every border point of *m_slopes*.

It ought to be said that we do not run **LevelData⟨FluxBox⟩**::*exchange* on either *m_slopes* or *m_coarsened_fine_data* in this code anywhere. But the **LevelData⟨FluxBox⟩**::*copyTo* method, which is invoked in chunk ⟨ Time Interpolation 27 ⟩, page 23, to transfer data from *a_old_coarse_data* and *a_new_coarse_data* to *m_coarsened_fine_data* does fill the ghost cells, and these are then used in interpolation onto the fine grid.

The user of this code is responsible for setting the interpolation radius sufficiently large so that all ghost cells of the fine grid are covered. This does not happen automatically, even though it could, because **LevelData⟨*T*⟩**::*ghostVect*( ) method can be used to return the number of ghost cells of the fine level field.

Once we have all items in place—the ghost cell margins and the disjoint box layout—we finally *define* *m_slopes* and *m_coarsened_fine_data*, whereupon we initialize all field components in the latter to −666.666. The reason why the "number of the beast" is used here is because we want a number that would stand out and be easy to recognize in case we have to debug the class or a program using it.

Here is the first time that we encounter a **DataIterator**. We will also encounter a **LayoutIterator** soon. There is one used in chunk ⟨ Collect coarse cells for interpolation 19 ⟩, page 16. A **LayoutIterator** returns boxes of a layout. They are naked, unadorned boxes. Unadorned by ghost cells. On the other hand **DataIterator** returns **FArrayBox**es or **FluxBox**es of the layout and these are defined on boxes that have been grown to incorporate ghost nodes. These boxes can be extracted from the fields. But **DataIterator** is an heir to **LayoutIterator** and can be applied to **BoxLayout**s as well, in which case it does the same as **LayoutIterator**.

The call to *m_coarsened_fine_data*[*dit*( )] returns a whole **FluxBox** defined over a box that's overgrown with ghost cells. The *setVal* method, sets all its components attached to all faces and in all cells of the overgrown box to −666.666.

14 ⟨ create private data structures 14 ⟩ ≡

```
assert(a_coarse_domain.checkPeriodic(a_crse_problem_domain));
DisjointBoxLayout coarsened_fine_domain;
coarsen(coarsened_fine_domain, a_fine_domain, m_ref_ratio);
const int coarse_slope_radius = (m_interp_radius + m_ref_ratio − 1)/m_ref_ratio;
const int coarse_ghost_radius = coarse_slope_radius + s_stencil_radius + 1;
const IntVect coarse_slope = coarse_slope_radius ∗ IntVect::Unit;
m_slopes.define(coarsened_fine_domain, a_num_comps, coarse_slope);
const IntVect coarse_ghost = coarse_ghost_radius ∗ IntVect::Unit;
```

```
m_coarsened_fine_data.define(coarsened_fine_domain, a_num_comps, coarse_ghost);
{
  DataIterator dit = coarsened_fine_domain.dataIterator( );
  for (dit.begin( ); dit.ok( ); ++dit) {
    m_coarsened_fine_data[dit( )].setVal(−666.666);
  }
}
```

This code is cited in chunks 7, 9, 13, and 21.

This code is used in chunk 11.

4.1.1 Loop over face directions

Now we are going to discuss chunk ⟨ loop over face directions 15 ⟩, first mentioned in chunk ⟨ Define the Border 11 ⟩, page 11.

This is the outermost iterative loop of *define*. It loops over the three directions that cell faces face (in 3D), $\boldsymbol{e}_x$, $\boldsymbol{e}_y$ and $\boldsymbol{e}_z$. For each face direction we are going to do separately what **PiecewiseLinearFillPatch**::*define*( ) does just once for cell centered data. This is the major difference between the two *define*s. It's basically like running **PiecewiseLinearFillPatch**::*define*( ) three times.

For each of these directions several other iterations will be carried out. The ultimate purpose of these is to identify all boundary nodes, both on the fine and on the coarse side, and classify the coarse side nodes, depending on whether we can calculate centered differences on them, or whether they are so far out that if we tried to calculate centered differences we would reach into "the void" while collecting data.

This may happen in one situation only, namely, if the outer border of the original coarse grid, i.e., the coarse/coarser border, is so close to the outer border of the fine grid, i.e., the fine/coarse border, that when we grow the boxes of the coarsened fine grid by adding ghost cells to them, the grown boxes protrude *beyond* the outer boundary of the coarse grid. More about this in chunk ⟨ Refine coarse cells sets 21 ⟩, page 18. Such cells will be put into one-sided slope sets, *m_coarse_lo_interp* and *m_coarse_hi_interp*, and all other cells will go into centered slope sets *m_coarse_centered_interp*.

Fine grid boundary cells are all put into one set, *m_fine_interp*. The sets will remain empty for in-land boxes that are away from the boundary.

All these collections and classifications are quite expensive. The loops are quadratic in the number of boxes per level, because we'll pick up a box and then for this box, we'll iterate over all other boxes of this level or of the coarse level performing various operations on the pairs of boxes so obtained. This is why for a static multigrid it is best to call *define* just once, at the beginning of the program, and not every time we need to fill fine region boundaries with data.

The first chunk in the loop, ⟨ Allocate grid point sets for each direction 16 ⟩, creates the sets—initially empty. The second chunk, ⟨ Make devices for testing periodic boundaries 17 ⟩, marks periodic boundaries, if there are any such. Finally, the third chunk, ⟨ Loop over boxes of the coarsened fine domain 18 ⟩, enters a yet another loop, this time over all boxes of the coarsened fine domain. So, the moment we get into it, we're going to look at the boxes, one after another, performing various operations on them, so as to generate and classify the sets of boundary nodes.

15 ⟨ loop over face directions 15 ⟩ ≡

```
for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
  ⟨ Allocate grid point sets for each direction 16 ⟩
  ⟨ Make devices for testing periodic boundaries 17 ⟩
  ⟨ Loop over boxes of the coarsened fine domain 18 ⟩
}
```

This code is cited in chunks 15 and 18.

This code is used in chunk 11.

¶ So, here we finally construct the set layouts, *m_fine_interp*, *m_coarse_centered_interp*, *m_coarse_lo_interp* and *m_coarse_hi_interp*, defined and discussed at some length in chunk ⟨ Protected Variables 7 ⟩, page 6. The

first one, *m_fine_interp* is defined over *a_fine_domain*, and the remaining set layouts are defined over *coarsened_fine_domain*, with the other index *dir* numbering directions in which field slopes will be evaluated.

Although we will only use slopes evaluated in the directions perpendicular to *faceDir*, we allocate space for all directions here, because this code derives from the cell-centered version, in which all directions were used.

16 ⟨ Allocate grid point sets for each direction 16 ⟩ ≡

```
m_fine_interp[faceDir].define(a_fine_domain);
for (int dir = 0; dir < SpaceDim; ++dir) {
  m_coarse_centered_interp[dir][faceDir].define(coarsened_fine_domain);
  m_coarse_lo_interp[dir][faceDir].define(coarsened_fine_domain);
  m_coarse_hi_interp[dir][faceDir].define(coarsened_fine_domain);
}
```

This code is cited in chunk 15.

This code is used in chunk 15.

¶ In this chunk we make two devices that will be used to check if boxes we'll compute on abut a periodic boundary. The devices are just boxes that are the same as *m_crse_problem_domain* or *fine_problem_domain* boxes with one exception. They are shortened by one row (both at the top and at the bottom) in the direction in which periodicity occurs. The tests will then be carried out by checking if a given box is *contained* within our device. If it is contained, it does not abut the boundary. If it is not contained, it means it does abut the periodic boundary and then we'll have to treat it specially.

These two devices will be used in chunks ⟨ Make correction for periodic boundary conditions 20 ⟩, page 17, ⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩, page 19, and ⟨ Collect fine cells for interpolation 23 ⟩, page 20.

17 ⟨ Make devices for testing periodic boundaries 17 ⟩ ≡

```
Box periodicTestBox(m_crse_problem_domain.domainBox());
if (m_crse_problem_domain.isPeriodic()) {
  for (int idir = 0; idir < SpaceDim; idir++) {
    if (m_crse_problem_domain.isPeriodic(idir)) periodicTestBox.grow(idir,-1);
  }
}
Box periodicFineTestBox(fine_problem_domain.domainBox());
if (m_crse_problem_domain.isPeriodic()) {
  for (int idir = 0; idir < SpaceDim; idir++) {
    if (m_crse_problem_domain.isPeriodic(idir)) periodicFineTestBox.grow(idir,-1);
  }
}
```

This code is cited in chunks 15 and 20.

This code is used in chunk 15.

4.1.2 Loop over boxes

Now we loop over all boxes of the *coarsened_fine_domain*. Recall that this loop is within the outermost loop of *define*, which is over the face directions *faceDir*, defined in chunk ⟨ loop over face directions 15 ⟩.

By now we have constructed (for the time being empty) sets that will eventually be filled with grid points onto which we will interpolate data (the fine grid) and from which we will collect data for interpolation (the coarse grid).

We have also made devices for testing for periodic boundaries on both the coarse and the fine levels.

The first operation we carry out within the box loop is the identification and collection of the *coarsened* fine grid cells that abut the boundary. This is where we are going to take data *from* for interpolation. We will assemble these points into a temporary local (i.e., associated with the box we're working on) set called *coarsened_fine_interp*.

We will then divide this set into three sets, depending on whether we can evaluate centered slopes or one-sided slopes for points contained in them. We refer to this step as *cell sets refinement*—perhaps a better term would be *cell sets division*, or *cell classification*.

All these operations will be performed on the *coarsened* fine grid, not on the original fine grid. They will result in filling *m_coarse_centered_interp*, *m_coarse_lo_interp* and *m_coarse_hi_interp* with points.

The last chunk in the box loop fills *m_fine_interp* with points. Observe that the disjoint box layouts for both the fine grid and the coarsened fine grid are identical. The only difference between the two grids is that the coarsened boxes are... coarsened. Consequently, we can use the same *dit*( ) for fetching fine grid and coarsened fine grid boxes.

18 ⟨ Loop over boxes of the coarsened fine domain 18 ⟩ ≡

```
DataIterator dit = coarsened_fine_domain.dataIterator( );
for (dit.begin( ); dit.ok( ); ++dit) {
  ⟨ Collect coarse cells for interpolation 19 ⟩
  ⟨ Refine coarse cells sets 21 ⟩
  ⟨ Collect fine cells for interpolation 23 ⟩
}
```

This code is cited in chunks 15 and 23.

This code is used in chunk 15.

Collect coarse cells for interpolation

So here is how we identify and collect *coarsened* fine grid boundary cells that will provide us with data for interpolation. Recall that this is the first operation we perform within the loop over the boxes of the *coarsened* fine grid box layout. The variable *dit*( ) selects a box of either the fine grid or the corresponding *coarsened* fine grid layout, *faceDir* selects the face direction for which we are doing all this.

We begin by picking an unadorned, ghost-cells free box from *a_fine_domain* that *dit*( ) points to and call it *fine_box*. Then we grow this box by *m_interp_radius* cells in *all* directions and *coarsen* it by *m_ref_ratio*. If any portion of the box produced this way protrudes out of the *m_crse_problem_domain* we chop it off. The final result is now called *coarsened_fine_facebox*.

Now, why do we do this? We are going to identify coarse/fine border cells by taking *every* box of the *coarsened_fine_domain* and subtracting it from the *coarsened_fine_facebox*. If the latter is somewhere in the middle of the grid, we'll end up subtracting all its points, so the result will be an empty set. But if the latter abuts a boundary, then the grown cells that protrude beyond the boundary will not be subtracted. These cells then will be saved in *coarsened_fine_interp*.

Now it is clear why *m_interp_radius* has to cover at least the ghost cells of *a_fine_data*. If it does not, the outermost ghost cells will not receive updates.

But before we get to do this, there is one subtlety we have to address.

Our data is face-mounted, not cell-center mounted. This means that for a given face direction we are going to have some data reside on the "left wall" of the box and on the "right wall" of the box. Yet cell-centered boxes of a **DisjointBoxLayout**, which are the ones we operate on here, do not account for this. The way we deal with it is that we include an additional layer of cells on the high side of the box, so as to cover that "right wall".

Here is how we go about it. First we call the **Box**::*surroundingNodes*( ) method. This method converts the box to `NODE` type in the direction specified, which adds the "right wall". Now we have both the "left wall" and the "right wall" in the box, but the box type has changed too, the nodes in the *faceDir* direction correspond to cell walls, not cell centers.

Now, this would be just what we want if the logic of the program was meticulously designed for it from the beginning. But this is not the case. We work here with the original cell-centered code of **PiecewiseLinearFillPatch** that has been merely tweaked to work with face centered data. So we are going to convert the `NODE` type of the box *back* to cell centers by shifting the nodes in the *faceDir* direction by one-half of the grid spacing. The effect of this operation is that we have simply added a layer of cells on the high side of the original box in the *faceDir* direction. We could have accomplished the same more economically by calling **Box**::*growHi*(*FaceDir*, 1).

We will have to repeat this operation on all other boxes that we're going to work with, including the ones we'll use to subtract grid points from this box.

Having produced the box we want, we get all the node points from it and put them in a set called *coarsened_fine_interp*. The reason for this is that the operation of subtracting one box from another cannot be carried out on boxes. This operation is well defined for *sets*, or for subtracting a box from a set, but not a box from a box.

So, we have all the points of our enlarged box with the "right wall" attached in the set. Now we enter a loop over all boxes of *coarsened_fine_domain*. For every box of the domain we stretch it in the *faceDir* direction by one layer of cells, so as to incorporate the "right wall" and then subtract it from *coarsened_fine_interp*.

In case a given box abuts a periodic boundary we have to treat it specially and this is deferred to chunk ⟨ Make correction for periodic boundary conditions 20 ⟩, which is discussed next.

19 ⟨ Collect coarse cells for interpolation 19 ⟩ ≡

```
const Box &fine_box = a_fine_domain[dit()];
Box coarsened_fine_facebox = coarsen(grow(fine_box, m_interp_radius), m_ref_ratio) & m_crse_problem_domain;
coarsened_fine_facebox.surroundingNodes(faceDir);
coarsened_fine_facebox.shiftHalf(faceDir, 1);
IntVectSet coarsened_fine_interp(coarsened_fine_facebox);
LayoutIterator other_lit = coarsened_fine_domain.layoutIterator();
for (other_lit.begin(); other_lit.ok(); ++other_lit) {
  Box other_coarsened_box = coarsened_fine_domain.get(other_lit());
  other_coarsened_box.surroundingNodes(faceDir);
  other_coarsened_box.shiftHalf(faceDir, 1);
  coarsened_fine_interp -= other_coarsened_box;
  ⟨ Make correction for periodic boundary conditions 20 ⟩
}
```

This code is cited in chunks 14, 21, 22, 23, and 35.

This code is used in chunk 18.

¶ So, how do we deal with periodic boundary conditions?

First, we check if there are any. This is what the *isPeriodic* method tells us. If the domain is periodic, we use the two devices we constructed in chunk ⟨ Make devices for testing periodic boundaries 17 ⟩, page 15. The devices were two domain-sized boxes that were shortened in the direction of periodicity both from the top and from the bottom by one layer of cells. So here we check if either *other_coarsened_box* or *coarsened_fine_facebox* are entirely contained within *periodicTestBox*. If both are contained, it means that neither touches the periodic boundary, and so we don't have to do anything.

If this is not the case, it means that either one or the other or both boxes touch a periodic boundary.

In this case we may have to go to the wrap-around region and subtract nodes from there.

Here is how we go about it. We extract the box from *m_crse_problem_domain*, which, as is defined in chunk ⟨ Protected Variables 7 ⟩, is a **ProblemDomain**—not a **BoxLayout**. The **Box** method *size*( ) returns an **IntVect**, which yields a size of the box in each direction, and we call it *shiftMult*.

The **ShiftIterator** *shiftIt* returns a *unit* **IntVect** for every direction in which *m_crse_problem_domain* is periodic, as has been defined in chunk ⟨ transfer data to private variables 12 ⟩, page 11. So here for every such periodic direction we define an **IntVect** called *shiftVect*, which points in the direction given by *shiftIt*, and the length of which is given by the size of the domain in this direction. The **Box** method *shift*( ) shifts the box by *shiftVect* cells, so that the box now wraps around the periodic domain. The cells of the wrapped around box are now subtracted from the *coarsened_fine_interp* set. After this operation is complete, the box is shifted back into its original position to prepare it for another shift in another periodic direction.

20 ⟨ Make correction for periodic boundary conditions 20 ⟩ ≡

```
if (m_crse_problem_domain.isPeriodic() ∧ ¬periodicTestBox.contains(other_coarsened_box) ∧
      ¬periodicTestBox.contains(coarsened_fine_facebox)) {
  IntVect shiftMult(m_crse_problem_domain.domainBox().size());
  Box shiftedBox(other_coarsened_box);
  for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
    IntVect shiftVect = shiftMult * shiftIt();
```

```
        shiftedBox.shift(shiftVect);
        coarsened_fine_interp -= shiftedBox;
        shiftedBox.shift(-shiftVect);
      }
    }
```

This code is cited in chunks 12, 17, 19, 22, and 23.

This code is used in chunk 19.

Refine coarse cell sets

Having collected all cells of the coarsened fine grid from which we are going to collect data for interpolation, we need to divide them depending on

1. the direction of slopes,

2. the ability to evaluate centered difference on them, if not put them in appropriate one-sided slope sets.

So we enter a loop over slope directions *dir*. These are not the same as *faceDir* directions. The latter describe the specific face directions for which we do all this. The former refer to the direction in which slopes (gradients) are going to be taken. Eventually we will be interested *only* in the directions that are *perpendicular* to *faceDir*, see, for example, chunk ⟨ Evaluation of Slopes 31 ⟩, page 26. But this code derives from a cell-centered version and prepares us for taking slopes in all directions, including *dir* = *faceDir*. This may come handy in future modifications.

As we enter this loop, the two indexes that we have inherited from the two higher level loops we're inside of, point to the box, *dit*( ), and to the face direction, *faceDir*.

The first thing we do is to extract the currently empty set that corresponds to this box, this *faceDir* and this slope direction, from *m_coarse_centered_interp*. We call this set *coarse_centered_interp* and we transfer all points collected in *coarse_fine_interp* in the preceding chunk ⟨ Collect coarse cells for interpolation 19 ⟩ to it. These were collected for *this* box and for *this faceDir*, but not specifically for this slope direction and we didn't check if all points were suitable for centered differences either. So *coarse_centered_interp* has too many points in it at present and we'll have to take some away.

In a similar manner we extract the currently empty sets that correspond to this slope direction, this face direction and this box from *m_coarse_lo_interp* and *m_coarse_hi_interp* and fill them both with points from *coarse_centered_interp*.

But having done this we shift all points in *coarse_lo_interp* "to the right" in the *dir* direction and all points in *coarse_hi_interp* "to the left" in the *dir* direction, by one cell.

Now, recall that we have oversized the ghost region of the coarsened fine grid in chunk ⟨ create private data structures 14 ⟩, page 12, so as to fit enough points outside of the *coarsened_fine_interp* set, reaching into the original coarse grid, to evaluate centered differences on *all* points of *coarsened_fine_interp*. The only time this is not going to happen is if the outer border of the original coarse grid, i.e., the coarse/coarser border, is so close to the outer border of the fine grid, i.e., the fine/coarse border, that when we produce the *grown* boxes of the coarsened fine grid, their ghost cells reach *beyond* the outer boundary of the coarse grid.[1]

So what we are going to do now is to subtract the boxes of the *original coarse grid,* not the coarsened fine grid, from these "left" or "right" shifted sets, *coarse_lo_interp* and *coarse_hi_interp*. Only the cells that "hang over the edge" will survive this subtraction, and these are the ones we want to have in *coarse_hi_interp* and in *coarse_lo_interp*. If the grid hierarchy has been constructed so that there are sufficiently wide margins between the outer coarse level grid and the outer fine level grid, these two sets will be always empty.

We could indeed use these two sets to raise flags in case we don't want to work with one-sided differences. In this case, we could go back to the place where we build the grid hierarchy and rebuild it with wider margins between grid boundaries.

Once we have carried out the subtraction, discussed in the next chunk ⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩, we shift the surviving points of *coarse_lo_interp* and *coarse_hi_interp* back into place, and having done so, we subtract both sets from *coarse_centered_interp*. So now, what's left in the latter are all those happy cells that can be used for centered differences in the *dir* direction.

[1] The weaver is indebted to Dan Martin for the clarification of this point.

21 ⟨ Refine coarse cells sets 21 ⟩ ≡

```
for (int dir = 0; dir < SpaceDim; ++dir) {
  IntVectSet &coarse_centered_interp = m_coarse_centered_interp[dir][faceDir][dit()];
  coarse_centered_interp = coarsened_fine_interp;
  IntVectSet &coarse_lo_interp = m_coarse_lo_interp[dir][faceDir][dit()];
  coarse_lo_interp = coarse_centered_interp;
  coarse_lo_interp.shift(BASISV(dir));
  IntVectSet &coarse_hi_interp = m_coarse_hi_interp[dir][faceDir][dit()];
  coarse_hi_interp = coarse_centered_interp;
  coarse_hi_interp.shift(-BASISV(dir));
  ⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩
  coarse_lo_interp.shift(-BASISV(dir));
  coarse_hi_interp.shift(BASISV(dir));
  coarse_centered_interp -= coarse_lo_interp;
  coarse_centered_interp -= coarse_hi_interp;
}
```

This code is cited in chunks 15 and 35.

This code is used in chunk 18.

¶ Here we subtract the coarse grid, *a_coarse_domain*, boxes from the two sets, *coarse_lo_interp* and *coarse_hi_interp*. The iterator *coarse_lit()* returns pointers to the actual boxes, which are then obtained with *get*(*coarse_lit()*) and *copied* onto *bx*. As we have done before, we stretch each *bx* in the *faceDir* direction by one layer of cells before we carry out the subtraction, so as to cover the "right wall" of the box, on which we have some data mounted.

If either *bx* or the *coarsened_fine_facebox* (which we have constructed and stretched in chunk ⟨ Collect coarse cells for interpolation 19 ⟩, this is the box we're still working on, the box pointed to by *dit()*) protrudes beyond a periodic boundary of the coarse grid, we wrap around the boundary and continue the subtraction on the other side, as we have done earlier in chunk ⟨ Make correction for periodic boundary conditions 20 ⟩. Then we shift the box *bx* back in place in preparation for wraping about the boundary in the next periodic direction.

22 ⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩ ≡

```
LayoutIterator coarse_lit = a_coarse_domain.layoutIterator();
for (coarse_lit.begin(); coarse_lit.ok(); ++coarse_lit) {
  Box bx = a_coarse_domain.get(coarse_lit());
  bx.surroundingNodes(faceDir);
  bx.shiftHalf(faceDir, 1);
  coarse_lo_interp -= bx;
  coarse_hi_interp -= bx;
  if (m_crse_problem_domain.isPeriodic() ∧ ¬periodicTestBox.contains(bx) ∧
      ¬periodicTestBox.contains(coarsened_fine_facebox)) {
    IntVect shiftMult(m_crse_problem_domain.domainBox().size());
    Box shiftedBox(bx);
    for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
      IntVect shiftVect = shiftMult * shiftIt();
      shiftedBox.shift(shiftVect);
      coarse_lo_interp -= shiftedBox;
      coarse_hi_interp -= shiftedBox;
      shiftedBox.shift(-shiftVect);
    }
  }
}
```

This code is cited in chunks 12, 17, and 21.

This code is used in chunk 21.

Collect fine cells for interpolation

So far we have collected coarse grid cells from which to interpolate data onto fine grid cells and we have divided them into three sets holding centered stencils, and one-sided stencils (two sets).

In this chunk we will collect cells of the fine grid onto which data will be interpolated.

Remember that in this part of the code we have the selected face direction in *faceDir* and the index of the selected box of the *coarsened_fine_domain* in *dit*( ). This index numbers both the boxes of the fine grid and of the *coarsened* fine grid.

We begin by picking up the, as yet empty, set of *m_fine_interp* that corresponds to this *faceDir* and to this box and give it a more tractable temporary name of *fine_interp*. Also, remember that right at the beginning of the loop over the boxes of the coarsed fine grid (see chunk ⟨ Loop over boxes of the coarsened fine domain 18 ⟩), in chunk ⟨ Collect coarse cells for interpolation 19 ⟩, we have picked up a *dit*( ) box from *a_fine_domain* and gave it a name of *fine_box*.

So here we make a copy of it first and call this copy *fine_faceBox*. Then we grow this box in *all* directions by *m_interp_radius* cells, but chop off anything that may stick out of the *fine_problem_domain*. Then we go again through the procedure of attaching the "right wall" of the box by stretching it in the *faceDir* direction by one layer of cells. Finally, we take all the cells out of that box and put them in the *fine_interp* set.

And so, at this point, the set contains all cells of the *fine_box*, plus all cells around it, to the width of *m_interp_radius*, plus an extra layer of cells on the *faceDir* face.

As we did before with the coarsened fine boxes, we will now subtract all *ungrown* boxes of the fine grid from this set. But before the subtraction, we stretch those boxes in the *faceDir* direction by one layer, to incorporate the "right wall". These boxes are *ungrown* because they are extracted from a **BoxLayout** and not from **LevelData**.

In case either the *fine_box* or the box that is being subtracted from it abuts a periodic domain boundary, we have to wrap around, and we do this as has been explained in chunk ⟨ Make correction for periodic boundary conditions 20 ⟩.

23 ⟨ Collect fine cells for interpolation 23 ⟩ ≡

```
IntVectSet &fine_interp = m_fine_interp[faceDir][dit()];
Box fine_faceBox(fine_box);
fine_faceBox.grow(m_interp_radius);
fine_faceBox &= fine_problem_domain;
fine_faceBox.surroundingNodes(faceDir);
fine_faceBox.shiftHalf(faceDir,1);
fine_interp.define(fine_faceBox);
LayoutIterator fine_lit = a_fine_domain.layoutIterator();
for (fine_lit.begin(); fine_lit.ok(); ++fine_lit) {
  Box bx = a_fine_domain.get(fine_lit());
  bx.surroundingNodes(faceDir);
  bx.shiftHalf(faceDir,1);
  fine_interp -= bx;
  if (fine_problem_domain.isPeriodic() ∧ ¬periodicFineTestBox.contains(fine_box) ∧
      ¬periodicTestBox.contains(bx)) {
    IntVect shiftMult(fine_problem_domain.domainBox().size());
    Box shiftedBox(bx);
    for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
      IntVect shiftVect = shiftMult * shiftIt();
      shiftedBox.shift(shiftVect);
      fine_interp -= shiftedBox;
      shiftedBox.shift(-shiftVect);
    }
  }
}
```

This code is cited in chunks 12 and 17.

This code is used in chunk 18.

4.2 Inquiry functions

This is a very small chunk: just the definition of the inquiry function *isDefined*( ). All the function does is to return the value of *m_is_defined*. This function isn't used anywhere within this source, because the class methods have direct access to *m_is_defined*.

24 ⟨ Inquiry functions 24 ⟩ ≡

```
bool PiecewiseLinearFillPatchFace::isDefined( ) const
{
  return (m_is_defined);
}
```

This code is cited in chunk 5.

This code is used in chunk 8.

4.3 Filling the Border

Filling the fine grid side of the coarse/fine border is done by function *fillInterp*. This function is essentially a wrapper that calls five functions that do the work. Its calling interface was discussed in chunk ⟨ Public Interfaces 5 ⟩, page 4. It is the only visible member of the class, the auxiliary functions it calls being all **protected**.

The body of the function begins with simple sanity checks. We ensure that the class and the coarse/fine border are fully defined and required data structures allocated.

Note that *assert* will do the checks and aborts *only* if the code is compiled with and linked against a `DEBUG` version of Chombo utilities. So, it is possible to call *fillInterp* on an *undefined* class in a non-`DEBUG` version of the code, in which case the code is bound to crash, unless it does not, and that is worse.

Next we ensure that the time interpolation coefficient *a_time_interp_coef* is restricted to $[0, 1]$.

Finally we *exchange* the designated components of *a_fine_data* between processes of the MPI pool. Observe that we do not run *a_old_coarse_data.exchange* and *a_new_coarse_data.exchange*. The user must ensure data integrity of the coarse level *before* calling *fillInterp*.

Now we are ready to carry out the interpolations.

First we time-interpolate the data between *a_old_coarse_data* and *a_new_coarse_data*. The time-interpolated data is written on the class protected variable *m_coarsened_fine_data*. This is the only time we look up the original coarse data. All following computations are carried out on *m_coarsened_fine_data*. The time interpolation function *timeInterp* is discussed in chunk ⟨ Time Interpolation 27 ⟩.

Next we carry out piecewise constant interpolation, i.e., we just copy a single value directly from the *coarsened* fine grid cell to all fine grid cells that live directly "under" the *coarsened* fine grid cell. This is done by function *fillConstantInterp*, which is discussed in chunk ⟨ Piecewise Constant Interpolation 29 ⟩. There is a little trick in this part of the code that ensures transfer of data between coarse and matching fine faces, as opposed to transfer of data from bulk to bulk of the cells as it's done in **PiecewiseLinearFillPatch**.

Finally, we add linear corrections to the piecewise constant interpolations. This is done first on the fine faces that overlap with the coarse faces of the *coarsened* fine grid—we call this a *tangent* correction—and then *between* the coarse faces of the *coarsened* fine grid—we call this a *normal* correction.

The tangent corrections are carried out by evaluating van Leer limited central differences for a given direction *dir* in the plane of the face first. This is done by function *computeSlopes* and the data is stored on *m_slopes*, separately for each face direction (remember that *m_slopes* is a **LevelData⟨FluxBox⟩**). Then function *incrementLinearInterpTangential*, which is discussed in chunk ⟨ Tangent Correction 35 ⟩, makes use of whatever it finds in *m_slopes* to add linear corrections to the numbers first written by *fillConstantInterp*.

Then we carry out a simple linear interpolation between the coarse faces of the *coarsened* fine grid to fill internal faces of the fine grid, i.e., fine grid faces that do not overlap with the *coarsened* fine grid faces. This is done by function *incrementLinearInterpNormal*, which is discussed in chunk ⟨ Normal Correction 37 ⟩.

And this is it.

25 ⟨ Fill the Border 25 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::fillInterp(LevelData⟨FluxBox⟩ &a_fine_data, const
        LevelData⟨FluxBox⟩ &a_old_coarse_data, const LevelData⟨FluxBox⟩ &a_new_coarse_data, Real
        a_time_interp_coef, int a_src_comp, int a_dest_comp, int a_num_comp)
{
  assert(m_is_defined);
  assert(a_time_interp_coef ≥ 0.);
  assert(a_time_interp_coef ≤ 1.);
  Interval fineComps(a_src_comp, a_src_comp + a_num_comp − 1);
  a_fine_data.exchange(fineComps);
  timeInterp(a_old_coarse_data, a_new_coarse_data, a_time_interp_coef, a_src_comp, a_dest_comp, a_num_comp);
  fillConstantInterp(a_fine_data, a_src_comp, a_dest_comp, a_num_comp);
  for (int dir = 0; dir < SpaceDim; ++dir) {
    computeSlopes(dir, a_src_comp, a_num_comp);
    incrementLinearInterpTangential(a_fine_data, dir, a_src_comp, a_dest_comp, a_num_comp);
  }
  incrementLinearInterpNormal(a_fine_data, a_src_comp, a_dest_comp, a_num_comp);
}
```

See also chunk 26.

This code is cited in chunks 5, 11, and 37.

This code is used in chunk 8.

¶ In the remaining part of this chunk we put place holders for the definitions of the auxiliary functions used by *fillInterp*. Observe that this chunk *appends* the code to the previous one.

26 ⟨ Fill the Border 25 ⟩ +≡
⟨ Time Interpolation 27 ⟩
⟨ Piecewise Constant Interpolation 29 ⟩
⟨ Evaluation of Slopes 31 ⟩
⟨ Tangent Correction 35 ⟩
⟨ Normal Correction 37 ⟩

4.3.1 Time Interpolation

The time interpolation function *timeInterp* takes two fields distributed over a box layout as its arguments, *a_old_coarse_data* and *a_new_coarse_data*. It will time-interpolate between them and write the result on the class protected field *m_coarsened_fine_data*. The time interpolation is linear and governed by a **Real** number *a_time_interp_coef*, which must be between 0 and 1. Zero (0) returns *a_old_coarse_data*, one (1) returns *a_new_coarse_data* and any number in between returns a linear combination of *a_new_coarse_data* and *a_old_coarse_data*.

This interpolation preserves divergence free property of both fields. Possible problems with divergence may show up when we get to space-interpolate the fields.

The interpolation may be carried out for some field components only, in which case the first component and the number of components to interpolate must be specified. These may be written at some other location in the destination field, in which case the first destination component must be specified.

The computation begins with a simple safety check. If both the old and the new fields are empty, we print an error message and abort through **MayDay**::*Error*. Otherwise, there are two simple cases that we can do without any computation. If *a_time_interp_coef* is 1 or if the old field is empty, then we can still return a sensible answer if the new field isn't empty. In this case we just copy the new field to *m_coarsened_fine_data*.

Similarly, if *a_time_interp_coef* is zero or the new field is empty, then we can still return a sensible answer if the old field isn't empty. In this case we just copy the old field to *m_coarsened_fine_data*.

Once we have dealt with these checks and simple cases, we're left with the actual interpolation, which is discussed in the next chunk, ⟨ Time-interpolate between old and new time slices 28 ⟩.

27 ⟨ Time Interpolation 27 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::timeInterp(const LevelData<FluxBox> &a_old_coarse_data, const
        LevelData<FluxBox> &a_new_coarse_data, Real a_time_interp_coef, int a_src_comp, int
        a_dest_comp, int a_num_comp)
{
  Interval src_interval(a_src_comp, a_src_comp + a_num_comp − 1);
  Interval dest_interval(a_dest_comp, a_dest_comp + a_num_comp − 1);
  if ((a_old_coarse_data.boxLayout( ).size( ) ≡ 0) ∧ (a_new_coarse_data.boxLayout( ).size( ) ≡ 0)) {
    MayDay::Error("PiecewiseLinearFillPatchFace::fillInterp: no old coarse dat\
        a and no new coarse data");
  }
  else if ((a_time_interp_coef ≡ 1.) ∨ (a_old_coarse_data.boxLayout( ).size( ) ≡ 0)) {
    a_new_coarse_data.copyTo(src_interval, m_coarsened_fine_data, dest_interval);
  }
  else if ((a_time_interp_coef ≡ 0.) ∨ (a_new_coarse_data.boxLayout( ).size( ) ≡ 0)) {
    a_old_coarse_data.copyTo(src_interval, m_coarsened_fine_data, dest_interval);
  }
  else {
    ⟨ Time-interpolate between old and new time slices 28 ⟩
```

```
    }
  }
```

This code is cited in chunks 5, 6, 14, 25, and 31.

This code is used in chunk 26.

¶ Now we get down to the actual time-interpolation. The computation here works as follows. First we copy the content of *a_new_coarse_data* to *m_coarsened_fine_data*, which is the final destination of this whole operation. We also copy the content of *a_old_coarse_data* to a new temporary field called *tmp_coarsened_fine_data*. Then we multiply the content of *m_coarsened_fine_data* by *a_time_interp_coef* and the content of *tmp_coarsened_fine_data* by $(1 - a\_time\_interp\_coef)$, and, finally, we add *tmp_coarsened_fine_data* to *m_coarsened_fine_data* and leave the result on the latter.

Some details—the temporary field *tmp_coarsened_fine_data* is created with the same number of components and ghost cells as *m_coarsened_fine_data* and on the same box layout. It is initialized, box-by-box, to −666.666. The computation is performed on a box-by-box basis, too, using Chombo data parallel operations such as the multiplication of all data in a box by a number and addition of data in another box to a target box. This is why we never have to extract the individual fields from **FluxBox**es, and we don't have to loop over the box cells either.

The *copyTo* method of the **LevelData** class copies all box and ghost cell data as well.

The resulting formula is

$$F(\lambda) = (1 - \lambda)F(0) + \lambda F(1)$$

where λ is *a_time_interp_coef*, and F is the field. It's easy to see that when λ is zero we get $F(0)$ and when λ is one we get $F(1)$.

28 ⟨ Time-interpolate between old and new time slices 28 ⟩ ≡

```
a_new_coarse_data.copyTo(src_interval, m_coarsened_fine_data, dest_interval);
const DisjointBoxLayout &coarsened_fine_layout = m_coarsened_fine_data.disjointBoxLayout();
LevelData<FluxBox> tmp_coarsened_fine_data(coarsened_fine_layout, m_coarsened_fine_data.nComp(),
    m_coarsened_fine_data.ghostVect());
{
  DataIterator dit = coarsened_fine_layout.dataIterator();
  for (dit.begin(); dit.ok(); ++dit) {
    tmp_coarsened_fine_data[dit()].setVal(-666.666);
  }
}
a_old_coarse_data.copyTo(src_interval, tmp_coarsened_fine_data, dest_interval);
DataIterator dit = coarsened_fine_layout.dataIterator();
for (dit.begin(); dit.ok(); ++dit) {
  FluxBox &coarsened_fine_fb = m_coarsened_fine_data[dit()];
  FluxBox &tmp_coarsened_fine_fb = tmp_coarsened_fine_data[dit()];
  for (int dir = 0; dir < SpaceDim; dir++) {
    coarsened_fine_fb[dir] *= a_time_interp_coef;
    tmp_coarsened_fine_fb[dir] *= (1. - a_time_interp_coef);
    coarsened_fine_fb[dir] += tmp_coarsened_fine_fb[dir];
  }
}
```

This code is cited in chunks 27 and 31.

This code is used in chunk 27.

4.3.2 Piecewise Constant Interpolation

This is a short function that just transfers data to the coarse/fine border region of *a_fine_data* from the overlapping region of *m_coarsened_fine_data*. Recall that by the time this function is called, *m_coarsened_fine_data* should have been filled with data that was time-interpolated by *timeInterp*( ).

The data in this function, *fillConstantInterp*, is not modified in any way yet. It is only copied to *a_fine_data*. The trick is to ensure that the data is moved between the right locations in both grids. What these locations are we are going to analyze closely in the next chunk, ⟨ Copy data between locations 30 ⟩.

The function takes the field *a_fine_data* as its argument. This field is going to be changed, on its boundary with the coarse region, by the function. The other arguments are the starting points for the source and destination components and the number of components.

First, the box layout is extracted from *a_fine_data* and a data iterator associated with it. The data iterator can be used both for *a_fine_data* boxes and for *m_coarsened_fine_data* boxes, since they overlap.

So, we enter a loop over the boxes of the layout. We extract a **FluxBox** from *a_fine_data* and call it *fine_flux*, then we also extract a corresponding **FluxBox** from *m_coarsened_fine_data* and call it *coarse_flux*. At this stage we enter a loop over the face directions of the box, since there is a different flux associated with each of them for both **FluxBox**es.

For each face direction *faceDir* we extract a flux from the *fine_flux* and from the *coarse_flux* and we call them *fine_fab* and *coarse_fab* correspondingly, "fab" being a Chombo moniker for a field of numbers (or columns of numbers) spanned over a box. Now we also get a set of points onto which we will interpolate data in a given *a_fine_data* box. The set of points lives in *m_fine_interp*[*faceDir*][*dit*( )]. We call the set *local_fine_interp* and we are going to iterate over all points of the set. The points will be returned by *ivsit*( ) inside the **for** loop, which, for every such point, locates and copies the corresponding number from the *coarse_fab*.

29 ⟨ Piecewise Constant Interpolation 29 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::fillConstantInterp(LevelData<FluxBox> &a_fine_data, int
        a_src_comp, int a_dest_comp, int a_num_comp) const
{
  DataIterator dit = a_fine_data.boxLayout().dataIterator();
  for (dit.begin(); dit.ok(); ++dit) {
    FluxBox &fine_flux = a_fine_data[dit()];
    const FluxBox &coarse_flux = m_coarsened_fine_data[dit()];
    for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
      FArrayBox &fine_fab = fine_flux[faceDir];
      const FArrayBox &coarse_fab = coarse_flux[faceDir];
      const IntVectSet &local_fine_interp = m_fine_interp[faceDir][dit()];
      IVSIterator ivsit(local_fine_interp);
      for (ivsit.begin(); ivsit.ok(); ++ivsit) {⟨ Copy data between locations 30 ⟩}
    }
  }
}
```

This code is cited in chunks 6, 7, 25, 31, 35, and 37.

This code is used in chunk 26.

¶ The fine grid point returned by the iterator is *ivsit*( ) and we call it *fine_iv*. Now, this point corresponds to the center of the fine grid cell. But our data does not live on cell centers. It lives on the walls.

If all the data was cell centered, then the algorithm would simply be *coarse_iv* = *coarsen*(*fine_iv*, *m_ref_ratio*), with *fine_iv* as returned by *ivsit*( ). For *m_ref_ratio* = 2 there would be four fine cells for each coarse cell, and each of these would get the value from the coarse cell that overlaps with it. Simple. This is what the standard Chombo function **PiecewiseLinearFillPatch**::*fillConstantInterp* does.

But for the face mounted data the code here does a little dance, which I don't think is really necessary.

Let us fix *m_ref_ratio* at 2 as above. Let *faceDir* correspond to left-right. The four fine cells that overlap with a single coarse cell can be divided into two left cells and two right cells.

Consider the left wall cells first. The algorithm in this chunk first shifts them to the left, so they end up under the left neighbor of the coarse cell. This is what *coarsen* returns. But now we shift this coarse cell back to the right and we end up with the original coarse cell. Then we shift the fine cells to the right as well. So, in effect, nothing will have changed. These two left cells will inherit a value that corresponds to the coarse cell they live in.

In this case the coarse cell value lives on the same face as the fine cell values, so this result is what we want

But now consider the right fine grid cells. When we shift them to the left the stay in the same *coarsened* fine grid cell. Then we shift the latter to the right and we again shift the former to the right. In effect the right fine grid cells get data not from their own *coarsened* fine grid cell, but from the one that's to the right of their cell.

Now, this is fine, because we are going to ignore this data in chunk ⟨ Normal Correction 37 ⟩ anyway. But then, what is the point of this dance in the first place? The *coarsened* fine grid face data would have been transferred correctly anyway.

30 ⟨ Copy data between locations 30 ⟩ ≡

```
IntVect fine_iv = ivsit( );
fine_iv.shift(faceDir, −1);
IntVect coarse_iv = coarsen(fine_iv, m_ref_ratio);
coarse_iv.shift(faceDir, +1);
fine_iv.shift(faceDir, +1);
int coarse_comp = a_src_comp;
int fine_comp = a_dest_comp;
for ( ; coarse_comp < a_src_comp + a_num_comp; ++fine_comp, ++coarse_comp)
  fine_fab(fine_iv, fine_comp) = coarse_fab(coarse_iv, coarse_comp);
```

This code is cited in chunk 29.

This code is used in chunk 29.

4.3.3 Evaluation of Slopes

Function *computeSlopes* evaluates van Leer limited gradients of fluxes associated with a given face in directions perpendicular to the direction of the face.

The gradients will then be used by *incrementLinearInterpTangential* to evaluate linear corrections to the constant values that have been written on the fine grid cells of the coarse/fine border by *fillConstantInterp* in chunk ⟨ Piecewise Constant Interpolation 29 ⟩.

The gradients of field components are evaluated on the coarse data, which by now should live in *m_coarsened_fine_data*, as it has been written on it by *timeInterp*, see chunks ⟨ Time Interpolation 27 ⟩ and ⟨ Time-interpolate between old and new time slices 28 ⟩.

The function takes a direction of the slope, an initial source component, and a number of components as its only three arguments. It does not take the destination component, because it does not do the writing on anything other than *m_slopes*.

The body of the function is a loop over face directions other than *a_dir*.

Before we enter the **if** (*faceDir* ≠ *a_dir*) clause, we initialize *m_slopes* to −666.666. Then we commence a loop over all boxes of the box layout *m_coarsened_fine_data* is defined on.

Once we have focused on a particular box *dit*( ) and a particular direction *faceDir*, we extract an **FArrayBox** (i.e., a flux through this face) from *m_coarsened_fine_data* and call it *data_fab*. We also extract an **FArrayBox** from *m_slopes* and call it *slope_fab*. Then we extract sets of cells that correspond to the slope direction *a_dir*, face direction *faceDir*, and box *dit*( ) from *m_coarse_centered_interp*, *m_coarse_lo_interp* and *m_coarse_hi_interp* and call them *local_centered_interp*, *local_lo_interp* and *local_hi_interp* respectively.

The evaluation of slopes is now carried out by three chunks. First we calculate van Leer limited central differences in ⟨ Evaluate van Leer limited central differences 32 ⟩, then low one-sided differences in ⟨ Evaluate low one-sided differences 33 ⟩, and finally high one-sided differences in ⟨ Evaluate high one-sided differences 34 ⟩.

31 ⟨ Evaluation of Slopes 31 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::computeSlopes(int a_dir, int a_src_comp, int a_num_comp)
{
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    {
      DataIterator dit = m_slopes.boxLayout( ).dataIterator( );
      for (dit.begin( ); dit.ok( ); ++dit) {
        m_slopes[dit( )][faceDir].setVal(−666.666);
      }
```

```
    }
    if (faceDir ≠ a_dir) {
      DataIterator dit = m_coarsened_fine_data.boxLayout().dataIterator();
      for (dit.begin(); dit.ok(); ++dit) {
        const FArrayBox &data_fab = m_coarsened_fine_data[dit()][faceDir];
        FArrayBox &slope_fab = m_slopes[dit()][faceDir];
        const IntVectSet &local_centered_interp = m_coarse_centered_interp[a_dir][faceDir][dit()];
        const IntVectSet &local_lo_interp = m_coarse_lo_interp[a_dir][faceDir][dit()];
        const IntVectSet &local_hi_interp = m_coarse_hi_interp[a_dir][faceDir][dit()];
        ⟨ Evaluate van Leer limited central differences 32 ⟩
        ⟨ Evaluate low one-sided differences 33 ⟩
        ⟨ Evaluate high one-sided differences 34 ⟩
      }
    }
  }
}
```

This code is cited in chunks 6, 21, 32, and 35.

This code is used in chunk 26.

¶ We begin the evaluation of centered differences by obtaining the **IVSIterator** for the *local_centered_interp* set. Now we loop over all points in the set, the *centered_ivsit*( ) returning an **IntVect**, which we call *iv*.

For each cell *iv*, we find its neighbors in the direction of the slope, *a_dir*, one "to the right", which we call *ivhi*, and one "to the left", which we call *ivlo*. And we can be certain that these neighbours exist and have valid data associated with them within *m_coarsened_fine_data*, because the points have been picked up from the *local_centered_interp* set.

Now we loop over all components specified when function *computeSlopes* was called in chunk ⟨ Evaluation of Slopes 31 ⟩. For each component we first evaluate the low slope $d_{\rm lo}$, then then high slope $d_{\rm hi}$, then take their average $(d_{\rm lo} + d_{\rm hi})/2$ and this is the central slope $d_{\rm center}$.

Now, the slope limiting works as follows. We first make the limited slope $d_{\rm lim}$ to be

$$d_{\rm lim} = 2\,\mathrm{minmod}\,(d_{\rm lo}, d_{\rm hi})\,,$$

which is zero if the signs of *dlo* and *dhi* differ, and 2 times the closer of the two to zero otherwise. Then we take again

$$\mathrm{minmod}\,(d_{\rm lim}, d_{\rm center})$$

If $d_{\rm lo}$ and $d_{\rm hi}$ are close and equal approximately to some d, then

$$2\,\mathrm{minmod}\,(d_{\rm lo}, d_{\rm hi}) \approx 2d$$

and the average of the two is d. Then $\mathrm{minmod}(2d, d)$ is always just d. So in this case we will always return the plain centered difference.

But if one of $d_{\rm lo}$ or $d_{\rm hi}$ is more than 3 times the other, then 2 times the lower one is less than the average of the two slopes, and in this case we return 2 times the smaller of the two slopes.

This is the van Leer central difference. Doing this prevents creation of artificial minima or maxima in the interpolated data. The factor 2 is not accidental or arbitrary.[2]

The value obtained this way is then transferred from $d_{\rm lim}$ to *slope_fab*(*iv*, *comp*).

32 ⟨ Evaluate van Leer limited central differences 32 ⟩ ≡

```
IVSIterator centered_ivsit(local_centered_interp);
for (centered_ivsit.begin(); centered_ivsit.ok(); ++centered_ivsit) {
  const IntVect &iv = centered_ivsit();
  const IntVect ivlo = iv - BASISV(a_dir);
  const IntVect ivhi = iv + BASISV(a_dir);
```

[2] The weaver owes again thanks to Dan Martin for pointing this.

```
    for (int comp = a_src_comp; comp < a_src_comp + a_num_comp; ++comp) {
      Real dlo = data_fab(iv, comp) − data_fab(ivlo, comp);
      Real dhi = data_fab(ivhi, comp) − data_fab(iv, comp);
      Real dcenter = .5 ∗ (dlo + dhi);
      Real dlim = 2. ∗ Min(Abs(dlo), Abs(dhi));
      if (dlo ∗ dhi < 0.) dlim = 0.;
      dlim = copysign(Min(Abs(dcenter), dlim), dcenter);
      slope_fab(iv, comp) = dlim;
    }
  }
```

This code is cited in chunks 9 and 31.

This code is used in chunk 31.

¶ There is less hocus pocus in this part of the code. Here we just pick up a point, called *iv*, from the *local_lo_interp* set. Then move one step "to the left" in the slope direction *a_dir*, and this new point is *ivlo*. Then we evaluate differences between field values at *iv* and *ivlo* for all components and store them in *slope_fab*.

We cannot make a step "to the right" in this case, because we'd fall off the edge... of the coarse grid.

33 ⟨ Evaluate low one-sided differences 33 ⟩ ≡

```
  IVSIterator lo_ivsit(local_lo_interp);
  for (lo_ivsit.begin(); lo_ivsit.ok(); ++lo_ivsit) {
    const IntVect &iv = lo_ivsit();
    const IntVect ivlo = iv − BASISV(a_dir);
    for (int comp = a_src_comp; comp < a_src_comp + a_num_comp; ++comp) {
      Real dlo = data_fab(iv, comp) − data_fab(ivlo, comp);
      slope_fab(iv, comp) = dlo;
    }
  }
```

This code is cited in chunks 31 and 34.

This code is used in chunk 31.

¶ Here we do the same as above, i.e., in chunk ⟨ Evaluate low one-sided differences 33 ⟩, but instead of moving "to the left", we move "to the right". The point to the right of *iv* is called *ivhi*. The differences between field values at *ivhi* and *iv* are evaluated for all components and stored on *slope_fab*.

Note that all the differences, central and one sided, go to *slope_fab*. Once they're in there, we no longer now which are which.

34 ⟨ Evaluate high one-sided differences 34 ⟩ ≡

```
  IVSIterator hi_ivsit(local_hi_interp);
  for (hi_ivsit.begin(); hi_ivsit.ok(); ++hi_ivsit) {
    const IntVect &iv = hi_ivsit();
    const IntVect ivhi = iv + BASISV(a_dir);
    for (int comp = a_src_comp; comp < a_src_comp + a_num_comp; ++comp) {
      Real dhi = data_fab(ivhi, comp) − data_fab(iv, comp);
      slope_fab(iv, comp) = dhi;
    }
  }
```

This code is cited in chunk 31.

This code is used in chunk 31.

4.3.4 Tangent Correction

Here we implement first linear corrections to the constant interpolants written on *a_fine_data* by *fillConstantInterp*, chunk ⟨ Piecewise Constant Interpolation 29 ⟩. The function takes *a_fine_data* as its argument, plus the slope direction *a_dir*, followed by the start source and destination components and the number of components to interpolate.

The body of the function is a large loop over face directions *faceDir* perpendicular to the slope direction *a_dir*. Within this loop we loop over all boxes of the *a_fine_data* box layout.

And so, for a given direction *faceDir*, different from *a_dir* and a given box pointed to by *dit*( ) we extract **FArrayBox**es from *m_slopes* and from *a_fine_data* and call them *slope_fab* and *fine_data_fab* respectively.

The *m_slopes* field would have been written on by *computeSlopes* in chunk ⟨ Evaluation of Slopes 31 ⟩ by now. This field contains van Leer limited centered differences and single side differences depending on the cells, which were collected and divided into *m_coarse_centered_interp*, *m_coarse_lo_interp* and *m_coarse_hi_interp* by *define* in chunks ⟨ Collect coarse cells for interpolation 19 ⟩ and ⟨ Refine coarse cells sets 21 ⟩. But by now we no longer need these sets, because we'll find the right data in the right places in *slope_fab*.

On the other hand we still need *m_fine_interp*, i.e., the field of sets of fine grid points onto which we are interpolating data from the coarse grid. We extract the set that corresponds to direction *faceDir* and box *dit*( ) and call it *fine_interp*. We are now going to iterate over all points of this set applying tangent linear corrections to data associated with them. If the set is empty, we don't do anything, of course.

35 ⟨ Tangent Correction 35 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::incrementLinearInterpTangential(LevelData⟨FluxBox⟩
        &a_fine_data, int a_dir, int a_src_comp, int a_dest_comp, int a_num_comp) const
{
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    if (faceDir ≠ a_dir) {
      DataIterator dit = a_fine_data.boxLayout().dataIterator();
      for (dit.begin(); dit.ok(); ++dit) {
        const FArrayBox &slope_fab = m_slopes[dit()][faceDir];
        FArrayBox &fine_data_fab = a_fine_data[dit()][faceDir];
        const IntVectSet &fine_interp = m_fine_interp[faceDir][dit()];
        IVSIterator ivsit(fine_interp);
        for (ivsit.begin(); ivsit.ok(); ++ivsit) {⟨ Apply tangent correction 36 ⟩}
      }
    }
  }
}
```

This code is cited in chunks 6, 7, 25, and 37.

This code is used in chunk 26.

¶ Finally we come to the very heart of this activity. The fine grid cell at which the correction is applied is called *fine_iv*, and *coarse_iv* is the coarse grid cell to which this fine grid cell belongs. This correction is applied within the plane of the face only.

Within the plane of the face the fine grid points are offset with respect to the coarse grid points the same way they are offset for cell-centered data. This is why this correction looks exactly like the correction made by function *incrementLinearInterp* in the **PiecewiseLinearFillPatch** class of standard Chombo.

The formula we want to apply here is as follows. Suppose *a_dir* points in the $\boldsymbol{e}_x$ direction.

Let x_f be the physical location of the fine grid point and x_c be the physical location of the corresponding coarse grid point. We find that $x_f = n_f \Delta x_f + x_{f0}$ and $x_c = n_c \Delta x_c + x_{c0}$, where $n_f/n_c = r$ is the refinement ratio. The correction to the value of field f at x_f is

$$f(x_f) = f(x_c) + \frac{\mathrm{d}f}{\mathrm{d}x}(x_f - x_c)$$

which is

$$f(x_f) = f(x_c) + \frac{\mathrm{d}f}{\mathrm{d}x}(n_f \Delta x_f + x_{f0} - n_c \Delta x_c - x_{c0})$$

The expression in the bracket can be transformed further as follows:

$$(n_f - n_c r)\Delta x_f + x_{f0} - x_{c0}$$

The origins of both grids are shifted with respect to each other by $(r-1) \times \Delta x_f/2$, with $x_{f0} < x_{c0}$, so

$$x_{f0} - x_{c0} = (1-r)\Delta x_f/2$$

Hence the multiplier of $\mathrm{d}f/\mathrm{d}x$ is

$$((1-r)/2 + (n_f - n_c r))\,\Delta x_f$$

The slope was evaluated for one coarse grid spacing, meaning that

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{\Delta f}{\Delta x_c}$$

Hence the correction term is

$$\frac{\Delta f}{\Delta x_c}\left((1-r)/2 + (n_f - n_c r)\right)\Delta x_f = \Delta f \frac{(1-r)/2 + (n_f - n_c r)}{r}$$

which yields

$$\left(-\frac{1}{2} + \frac{n_f - n_c r + 1/2}{r}\right)\Delta f$$

Now let us have a look at the code. We evaluate $n_f - n_c r$ first and this goes into *offset*. Then we evaluate the whole multiplier and this goes into *interp_coef*. Now we enter the loop over the field components and add the corresponding $(x_f - x_c)\mathrm{d}f/\mathrm{d}x$ corrections to the values in the *fine_data_fab* at this point.

Observe that even though we have been emphasizing that this is done on the fine grid faces that overlap with coarse grid faces, the code, in fact, does it everywhere, sic! The point is that in the next chunk, we will *overwrite* completely the values on fine grid faces that do not overlap with the coarse grid faces.

There is a lot of redundant computation in this code.

36 ⟨ Apply tangent correction 36 ⟩ ≡

```
const IntVect &fine_iv = ivsit( );
const IntVect coarse_iv = coarsen(fine_iv, m_ref_ratio);
const int offset = fine_iv[a_dir] − m_ref_ratio ∗ coarse_iv[a_dir];
Real interp_coef = −.5 + (offset + .5)/m_ref_ratio;
int coarse_comp = a_src_comp;
int fine_comp = a_dest_comp;
for ( ; coarse_comp < a_src_comp + a_num_comp; ++coarse_comp, ++fine_comp) {
  fine_data_fab(fine_iv, fine_comp) += interp_coef ∗ slope_fab(coarse_iv, coarse_comp);
}
```

This code is cited in chunk 37.

This code is used in chunk 35.

4.3.5 Normal Correction

This chunk implements linear correction to the interpolation in the direction that's normal to the face. Recall that function *incrementLinearInterpNormal* is called *outside* the **for** loop of *fillInterp* (see chunk ⟨ Fill the Border 25 ⟩). It takes *a_fine_data*, starting components on both the source and destination side and the number of components as its only arguments.

Both the source and the destination data live on *a_fine_data*, because we have already transferred data to it from *m_coarsened_fine_data* in chunk ⟨ Piecewise Constant Interpolation 29 ⟩ and applied sideways corrections to it in chunks ⟨ Tangent Correction 35 ⟩ and ⟨ Apply tangent correction 36 ⟩. The normal correction here basically recomputes values on faces that do not overlap with coarse faces by averaging them between the coarse grid faces.

The body of the function is a loop over face directions. For each direction we construct an **IntVect** that points in this direction and its length is equal to the refinement ratio *m_ref_ratio*. Then we enter a loop over all boxes of the *a_fine_data* box layout.

Within this loop we extract the **FArrayBox** for the face and the box from *a_fine_data* and call it *fine_data_fab*. Then we extract the set of fine interpolation points for this box and this face direction and call it *fine_interpIVS*. And then we get to loop over these points. Of course, if the set is empty, we don't do anything, so whatever happens next happens to border boxes only.

For each grid point in a border box, we call this point *fine_iv*, we extract the coordinate of this point in the *faceDir* direction and call it *loEdgeComp*. Then we check if this coordinate corresponds to a face of the coarse grid. If the point is on the face of the coarse grid, then *loEdgeComp* divides by *m_ref_ratio*. So if we divide it by *m_ref_ratio* and then multiply by it, we should get back its original value. We compare the value obtained against the original value, which is re-retrieved from *fine_iv* and if it checks, i.e., if the point lies on the coarse grid face ... we do nothing.

If the point does not lie on the coarse grid face, then we interpolate.

37 ⟨ Normal Correction 37 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::incrementLinearInterpNormal(LevelData<FluxBox>
        &a_fine_data, int a_src_comp, int a_dest_comp, int a_num_comp) const
{
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    IntVect hiShift(IntVect::TheZeroVector());
    hiShift.setVal(faceDir, m_ref_ratio);
    DataIterator dit = a_fine_data.dataIterator();
    for (dit.reset(); dit.ok(); ++dit) {
      FArrayBox &fine_data_fab = a_fine_data[dit()][faceDir];
      const IntVectSet &fine_interpIVS = m_fine_interp[faceDir][dit()];
      IVSIterator ivsit(fine_interpIVS);
      for (ivsit.begin(); ivsit.ok(); ++ivsit) {
        const IntVect &fine_iv = ivsit();
        int loEdgeComp = fine_iv[faceDir];
        loEdgeComp = m_ref_ratio * (loEdgeComp/m_ref_ratio);
        if (loEdgeComp ≠ fine_iv[faceDir]) {⟨ Interpolate between faces 38 ⟩}
      }
    }
  }
}
```

This code is cited in chunks 6, 25, and 30.

This code is used in chunk 26.

¶ So, now we are looking at a point that is between the coarse grid faces. The vector called *loVect* corresponds to the low coarse grid face and *hiVect* corresponds to the high grid face. Our point *fine_iv* is between them. The distance, in *coarse* grid cells, between *fine_iv* and the low coarse grid face is *fraction*. It is evaluated initially in fine grid cells, but is then scaled to coarse grid cells when it gets divided by *m_ref_ratio*. Let us call this fraction λ, let the position of the low coarse grid face be $x_{\rm lo}$ and the position of the high coarse grid face by $x_{\rm hi}$. Then

$$\lambda = \frac{x - x_{\rm lo}}{x_{\rm hi} - x_{\rm lo}}.$$

Now we enter the loop over the components of the field and implement the following calculation at x. Let $f(x_{\rm lo})$ and $f(x_{\rm hi})$ be field values at the low and high faces respectively. Then

$$f(x) = \lambda f(x_{\rm hi}) + (1 - \lambda) f(x_{\rm lo}).$$

Observe that we do not add a correction to $f(x)$ here. We overwrite completely whatever value may have been placed in *fine_data_fab* by *fillConstantInterp*, and later also by *incrementLinearInterpTangential*, with a weighed average of $f(x_{\rm lo})$ and $f(x_{\rm hi})$.

38 ⟨ Interpolate between faces 38 ⟩ ≡

```
IntVect loVect(fine_iv);
loVect.setVal(faceDir, loEdgeComp);
IntVect hiVect = loVect + hiShift;
Real fraction = fine_iv[faceDir] - loEdgeComp;
fraction = fraction/m_ref_ratio;
for (int comp = a_dest_comp; comp < a_dest_comp + a_num_comp; ++comp) {
  fine_data_fab(fine_iv, comp) = (1.0 - fraction) * fine_data_fab(loVect, comp);
  fine_data_fab(fine_iv, comp) += fraction * fine_data_fab(hiVect, comp);
}
```

This code is cited in chunk 5.

This code is used in chunk 37.

4.4 Debugging Utilities

There isn't much here in terms of debugging. But we have a simple method that lets us print all the sets constructed by *define*. The function loops over the face directions, then over the boxes of the *m_fine_interp* box layout and prints the *m_fine_interp* set for each box and face direction. Then it enters another internal loop, over the slope directions, and prints *m_coarse_centered_interp*, *m_coarse_lo_interp* and *m_coarse_hi_interp* for each slope direction, face direction and box.

All output is on *cout*, not on *pout*( ). So this function cannot be used in a parallel context.

It would be a good idea to expand on this a little and make this function also write Gnuplot data files for display of the actual grid points collected in each of the sets. Another iteration could make it write a Chombo HDF5 file with the same data.

39 ⟨ Debugging Utilities 39 ⟩ ≡

```
void PiecewiseLinearFillPatchFace::printIntVectSets( ) const
{
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    cout << "face direction = " << faceDir << endl;
    DataIterator lit = m_fine_interp[faceDir].boxLayout( ).dataIterator( );
    for (lit.begin( ); lit.ok( ); ++lit) {
      cout << "grid " << lit( ).intCode( ) << ": " << endl;
      cout << "fine ivs" << endl;
      cout << m_fine_interp[faceDir][lit( )] << endl;
      for (int dir = 0; dir < SpaceDim; ++dir) {
        cout << "coarse centered ivs [" << dir << "]: " << endl;
        cout << m_coarse_centered_interp[dir][faceDir][lit( )] << endl;
        cout << "coarse lo ivs [" << dir << "]: " << endl;
        cout << m_coarse_lo_interp[dir][faceDir][lit( )] << endl;
        cout << "coarse hi ivs [" << dir << "]: " << endl;
        cout << m_coarse_hi_interp[dir][faceDir][lit( )] << endl;
      }
    }
  }
}
```

This code is cited in chunk 5.

This code is used in chunk 8.

¶

Index

Here is a list of the identifiers used, and the chunks where they appear. Underlined entries indicate the place of definition.

List of Refinements

⟨ Allocate grid point sets for each direction 16 ⟩ Cited in chunk 15. Used in chunk 15.
⟨ Apply tangent correction 36 ⟩ Cited in chunk 37. Used in chunk 35.
⟨ CPP File Includes 9 ⟩ Cited in chunks 7 and 14. Used in chunk 8.
⟨ Collect coarse cells for interpolation 19 ⟩ Cited in chunks 14, 21, 22, 23, and 35. Used in chunk 18.
⟨ Collect fine cells for interpolation 23 ⟩ Cited in chunks 12 and 17. Used in chunk 18.
⟨ Copy data between locations 30 ⟩ Cited in chunk 29. Used in chunk 29.
⟨ Debugging Utilities 39 ⟩ Cited in chunk 5. Used in chunk 8.
⟨ Define the Border 11 ⟩ Cited in chunks 14 and 15. Used in chunk 8.
⟨ Evaluate high one-sided differences 34 ⟩ Cited in chunk 31. Used in chunk 31.
⟨ Evaluate low one-sided differences 33 ⟩ Cited in chunks 31 and 34. Used in chunk 31.
⟨ Evaluate van Leer limited central differences 32 ⟩ Cited in chunks 9 and 31. Used in chunk 31.
⟨ Evaluation of Slopes 31 ⟩ Cited in chunks 6, 21, 32, and 35. Used in chunk 26.
⟨ Fill the Border 25, 26 ⟩ Cited in chunks 5, 11, and 37. Used in chunk 8.
⟨ Includes 4 ⟩ Used in chunk 3.
⟨ Inquiry functions 24 ⟩ Cited in chunk 5. Used in chunk 8.
⟨ Interpolate between faces 38 ⟩ Cited in chunk 5. Used in chunk 37.
⟨ Loop over boxes of the coarsened fine domain 18 ⟩ Cited in chunks 15 and 23. Used in chunk 15.
⟨ Make correction for periodic boundary conditions 20 ⟩ Cited in chunks 12, 17, 19, 22, and 23. Used in chunk 19.
⟨ Make devices for testing periodic boundaries 17 ⟩ Cited in chunks 15 and 20. Used in chunk 15.
⟨ Normal Correction 37 ⟩ Cited in chunks 6, 25, and 30. Used in chunk 26.
⟨ Piecewise Constant Interpolation 29 ⟩ Cited in chunks 6, 7, 25, 31, 35, and 37. Used in chunk 26.
⟨ `Prolongate.H` 3 ⟩
⟨ Protected Interfaces 6 ⟩ Cited in chunk 5. Used in chunk 3.
⟨ Protected Variables 7 ⟩ Cited in chunks 9, 11, 16, and 20. Used in chunk 3.
⟨ Public Interfaces 5 ⟩ Cited in chunks 10, 11, and 25. Used in chunk 3.
⟨ Refine coarse cells sets 21 ⟩ Cited in chunks 15 and 35. Used in chunk 18.
⟨ Subtract coarse domain boxes from one sided stencils 22 ⟩ Cited in chunks 12, 17, and 21. Used in chunk 21.
⟨ Tangent Correction 35 ⟩ Cited in chunks 6, 7, 25, and 37. Used in chunk 26.
⟨ Time Interpolation 27 ⟩ Cited in chunks 5, 6, 14, 25, and 31. Used in chunk 26.
⟨ Time-interpolate between old and new time slices 28 ⟩ Cited in chunks 27 and 31. Used in chunk 27.
⟨ Wrappers for Define 10 ⟩ Used in chunk 8.
⟨ create private data structures 14 ⟩ Cited in chunks 7, 9, 13, and 21. Used in chunk 11.
⟨ loop over face directions 15 ⟩ Cited in chunks 15 and 18. Used in chunk 11.
⟨ perform sanity checks 13 ⟩ Cited in chunk 14. Used in chunk 11.
⟨ transfer data to private variables 12 ⟩ Cited in chunk 20. Used in chunk 11.